

Chimera Virtual Data System

Version 1.2

User Guide

[Version 41]

December 11, 2003

Send comments to: chimera-support@griphyn.org

Contents

1	Overview	3
2	Functionality	3
2.1	Release Overview	3
2.2	Workflow of the GVDS	5
2.3	Not included in this release	6
3	Packaging	6
3.1	Release Contents	6
3.2	Release Directory Structure	7
3.3	Platform Support	8
3.4	Requirements for the catalog and job submission site(s)	8
3.5	Run-time environment requirements	8
4	Execution environment	9
4.1	Overview	9
4.2	Condor setup	11
4.3	Properties	12
4.4	Virtual Data Catalog execution environment	16
4.4.1	The VDC database schema layer	17
4.4.2	The Provenance Tracking Catalog (PTC) schema	19
4.4.3	VDC and PTC combined database drivers	19
4.4.4	PostgreSQL 7.3	20
4.4.5	MySQL 4.0	23
4.4.6	Other database engines	25
4.5	DAG Execution - Grid Environment	25
4.6	Profiles	27
4.7	Security	28
4.8	File naming, Data Transfer, and Replication	28
4.9	The Transformation Catalog	31
4.9.1	The Transfer Mechanism	32
4.10	Replica Registration	34
4.10.1	Using the Globus Replica Catalog	34
4.10.2	Using the Replica Location Service	34
4.10.3	Installing the Replica Location Service	35
4.10.4	Setting up your own RLS server	36
4.11	Pool Configuration using MDS	37
4.11.1	XML Based pool configuration file	37
4.11.2	Setup of an MDS-driven configuration	38
4.11.3	Generation of the pool config file from MDS	39
5	Command Line Toolkit	40
5.1	Conversion between VDLt and VDLx	41

5.2	Creating, adding, and updating definitions	42
5.3	Deleting definitions	42
5.4	Searching, listing and dumping	43
5.5	Producing an abstract DAG in XML (“DAX”)	43
5.6	Running the shell planner	44
5.7	Supplementing the remote start-up	45
5.8	Obtaining remote job failure status and the provenance trail	45
5.8.1	Failing workflows on remote job failure	45
5.9	Requesting a Concrete DAG	45
5.9.1	Running the Pegasus concrete planner	45
5.9.2	Naming Conventions for the Condor files	46
5.9.3	Running the concrete DAG	46
6	Examples	47
6.1	The Shell Planner	47
7	Documentation and some Frequently Asked Questions	48
7.1	Before you shout help	49
7.1.1	Verify that you are permitted on the remote site	49
7.1.2	Verify that the services are up, running and accessible	50
7.1.3	Verify that you can run simple jobs	50
7.1.4	Verify that you can transfer files	51
7.1.5	Verify that you can run local Condor jobs	51
7.1.6	Verify that you can run Condor-G jobs	52
7.1.7	Verify that you can run DAGMan in its full beauty	54
7.2	If nothing else helps	54
8	Test Scripts	55
8.1	Verifying Your Foundation	55
8.2	Running VDS samples	56
8.2.1	Hello World	57
8.2.2	A More Complex Example – A Black Diamond	58
9	Appendix I: Glossary of Terms	60
10	Appendix II: Primer to the VDLt textual input language	61
10.1	Overview	61
10.2	The Transformation TR	62
10.3	The Derivation DV	62
10.4	The Other Elements	63
10.5	A Diamond Example	64
11	Appendix III: Managing the replica catalogs	65
11.1	The Globus-2 Replica Catalog	65
11.1.1	Creating the setup_rc file	65
11.1.2	Java Client to access the replica catalog servers	65
11.1.3	Creating a collection	66
11.1.4	Creating locations	66
11.1.5	Adding filenames to existing location	66
11.1.6	Listing the filenames associated with the collection	66
11.1.7	Listing filenames with a particular location	67
11.1.8	Listing all the locations associated with collection	67
11.1.9	Listing all the locations including their attributes	67
11.1.10	Deleting filenames	67
11.2	The Replica Location Service	67
11.2.1	Defining the pool attribute to be associated with the PFN	67
11.2.2	Adding a single LFN to PFN mappings	67

11.2.3	Subsequent additions for the same existing LFN.....	67
11.2.4	Adding the pool attribute value for each PFN	67
11.2.5	Adding the entries using bulk mode.	68
11.2.6	Querying a LRC	68
11.2.7	Querying a RLI	68
11.2.8	Deleting a LFN from a LRC	68
11.2.9	Making a LRC update to a RLI.....	68
11.2.10	Making a LRC force update to a RLI	68
12	Appendix IV: Miscellaneous Issues	69
12.1	Namespaces	69
12.2	Special properties for future releases.....	69
12.3	Tests.....	69
13	Known Issues	69
13.1	Document Issues to Address.....	70

1 Overview

The Chimera Virtual Data System (VDS) provides a means to describe a desired data product and to produce it in the Grid environment. The VDS provides a catalog that can be used by application environments to describe a set of application programs (“transformations”), and then track all the data files produced by executing those applications (“derivations”). The VDS contains the “recipe” to produce a given logical file, and on request produces a directed acyclic graph of program execution steps which will produce the file. These steps are translated into an executable DAGMan DAG by the Pegasus Planner, which is also included in this release and described in these notes.

Pegasus, which stands for Planning for Execution in Grids takes the recipe in the form of an abstract DAG and based on the available resources and replica locations creates a concrete DAG that will produce the requested file. The concrete DAG indicates which resources need to be used to execute the tasks in the graph and describes the necessary data movement. Additionally, the final and/or intermediate data products can be registered for future reuse.

This user guide describes the first Beta release of the Virtual Data System that is intended for use within the GriPhyN project. It may be used, but with less guaranteed support, for evaluation by other interested parties. It is intended to give prospective users a preview of the release and to solicit feedback on the release’s suitability for applications. It covers only the externally visible features of the release, not the details of the system’s internal design.

This VDS “preview release” does not yet contain all features necessary for production application deployment, but it will serve the valuable purpose of providing a hands-on laboratory in which to try virtual data applications.

Please refer to section 9 “Appendix I: Glossary of Terms” for details on the acronyms.

2 Functionality

2.1 Release Overview

This release implements the textual and XML versions (VDLt and VDLx) of the Virtual Data Language version 1.1. This language is defined in section 10 of this document. A separate document contains a tutorial and language reference for VDLt.

The core components of this release are coded in Java. The functionality of these components is also made available through a set of command-line applications. These applications perform creation of transformation and derivations, produce abstract dags, and produce concrete dags, ready to be executed by `condor_submit_dag`.

VDL definitions (transformations and derivations) are stored as a file of XML elements for persistence. (These will be stored in a database in the next release).

A sequence of commands to define and execute a simple derivation `d1` stored in the file “`vdltdefs`” would be:

```

JAVA_HOME=/usr/lib/java; export JAVA_HOME # match your own env.
VDS_HOME=/u/me/vds-1.0b4; export VDS_HOME # dito
. $VDS_HOME/setup-user-env.sh
vdl2vdlx vdldefs vdlxdefs
updatevdc vdlxdefs # Merges mods from vdlxdefs into persistence storage
searchvdc -t derivation -i d1 # produces output, not shown
gendax -l test -i d1 -o DAX-file
gencdag --dir dagdir --p ufl --o uw --dax DAX-file
cd dagdir; condor_submit_dag test.dag

```

Note that this sequence is for Bourne-shell users, and that that you must first set up an `etc/properties` file, or your user properties file, as explained in section 4.2.

After this sequence of commands is executed, the “concrete DAG” generated in `dagdir` will contain:

- all steps needed to move files from the storage elements where they currently exist to the storage element where they are accessible to the execution nodes of the execution site.
- Steps to execute all derivations in the DAG
- steps to return data products to a designated destination storage element specified by the `-out` argument. If no output pool (`--o` option to `gencdag`) is specified, the data remains at the computation pool (`--p` option to `gencdag`).
- steps to catalog all output data products in a replica catalog or Replica Location Service depending upon the replica mode specified in the properties file, as explained in section 4.2. The nodes for registering are added if the user specifies an output pool, where he wants the materialized data products to be transferred to.

The last command, “`condir_submit_dag`”, submits the DAG to Condor for execution.

The user must tell `gencdag` the site (pool) at which to execute the concrete DAG (via the `--p` option). The pool must be chosen from the list of pools in the `pools.config` file.

The Pegasus concrete planner (`gencdag`) converts an abstract DAG (DAX) produced by `gendax` into a concrete DAG, which can be executed in a grid environment consisting of Condor pools. To construct the concrete dag, the planner uses the services of Replica Catalog/Replica Location Service and a Transformation Catalog (which is currently a simple text file). A set of tools are provided that enable the user to seed a replica catalog with initial data locations. *Pegasus* allows for the delivery of data to a particular location as requested by the user. The `gencdag` command generates a DAG consisting of jobs, which are executed on the grid using *Condor-G*, *Condor*, and the *Globus Toolkit*. These jobs may involve the materialization of requested files and their dependencies, the transfer of existing files from other locations and the registration of all the materialized files in the Replica Catalog.

2.2 Workflow of the GVDS

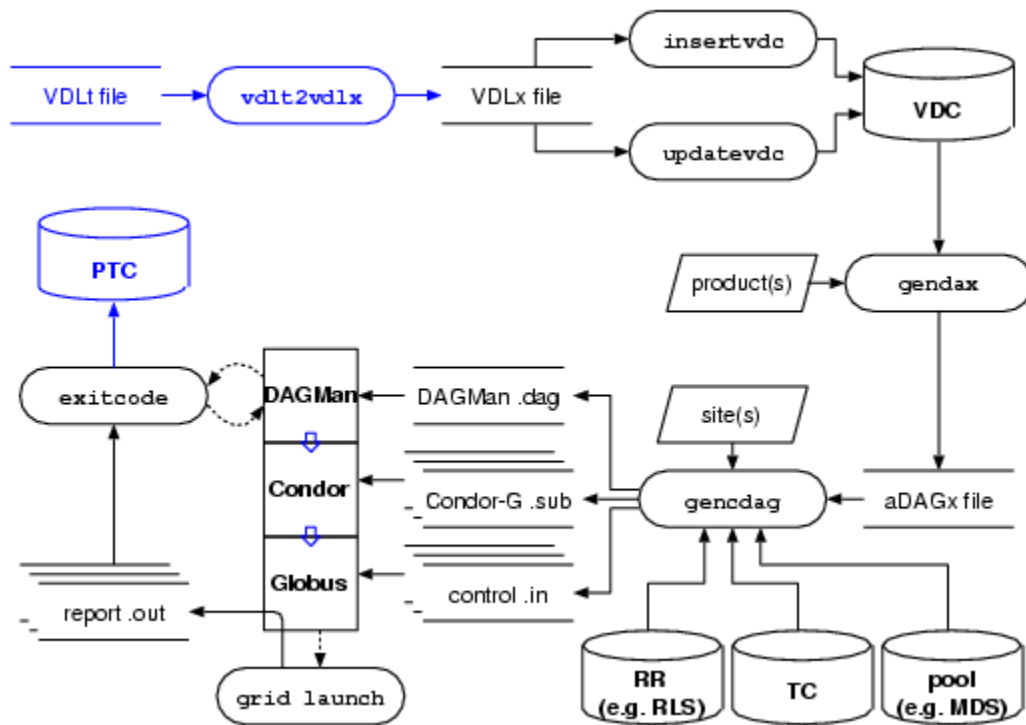


Figure 1: Overview of the GVDS workflow.

Figure 1 shows an overview over the steps involved to run the GriPhyN Virtual Data System in a grid environment. The parallel lines enclose data, usually files. The oval processes contain the name of the program to be executed. The disk-shaped items denote databases. In the parallelogram, essential input is being portrayed. Square boxes denote pieces that are outside the Virtual Data System. Solid lines are data flow, dashed lines control flow.

While the textual input language VDLt is useful for human interaction, the *lingua franca* of the virtual data system is XML. The `vdl2vdx` application translates VDLt into VDLx. For larger population of the virtual data space, we highly recommend using a scripting approach, which produces VDLx directly. Thus, this approach is marked in blue.

The virtual data definitions need to be added into the VDC before they are useful. You can either insert them, which will not allow duplicates, or update the VDC, which will overwrite an existing version of the VDC with the provided version. Usually, insertion is faster than updating.

To obtain a virtual data product, the `gendax` program will search the virtual data space that is contained in the VDC. Providing the name of a data product, either a derivation identifier, or a logical filename, the complete workflow necessary to produce the requested piece of data will be generated. The resulting *abstract DAG in XML (DAX)* file contains the name of all logical files that are part of the workflow, the job description, and the job interdependencies. The file constitutes the logical plan, and all elements, transformations and filenames, are logical.

Any concrete planner takes the abstract workflow, and turns it into a concrete workflow that can be executed, employing information from the transformation catalog, which maps logical transformations into physical applications. Another part of the process is to augment the workflow with file management jobs: stage-in, stage-out and replica registration. The concrete planning process also prunes the graph by omitting computation jobs for results that already exist, using the provided replica registry. In the grid environment,

the use of RLS is recommended. The pool configuration catalog contains the necessary information to produce jobs that are grid-capable.

The product of `gencdag` is a plan in several files. The dot-dag file is the control file for the Condor-DAGMan workflow manager. The dot-sub files contain job descriptions for the Condor-G system, which is part of Condor. Also, a number of control files for remote jobs may be generated as dot-in files.

A concrete workflow is started by submitting it to Condor DAGMan. DAGMan itself is a Condor job, and will also schedule Condor jobs and Condor-G jobs, based on the submit files and the control-flow dependencies. While Condor jobs run local to the submit host, Condor-G jobs run via Globus in the Grid.

Grid jobs and local jobs alike are best started from a grid launcher. The provided launcher is `kickstart`, and there exists an alternative implementation as Condor `gridshell`. These launchers start the true application, and capture certain information like exit code and resource consumption into a report. These reports can be read by a DAGMan post-script file `exitcode`, which provides DAGMan with information about remote job failure, and can also write a provenance record into the Provenance Tracking Catalog (PTC).

2.3 Not included in this release

In the interest of making this first VDS release simple and available as soon as possible, it is being provided without the features listed here. These will be supplied as soon as possible in subsequent releases.

- There is no VDS server (VDL functions all execute on the local host).
- The interfaces to the VDS are Java classes and command-line scripts – no other language bindings are provided in this release.
- Only Linux is supported. (Compatibility with other UNIX systems such as Solaris is likely, but will not be tested in this release. The Java code should also run under Windows but, likewise, that platform will not be tested or supported in this release.)

3 Packaging

3.1 Release Contents

VDS-1.2 will be packaged as:

- A set of Java classes
- A set of supporting Java libraries
- Shell scripts and command line tools (most or all of which are written in Java with simple shell wrappers).
- XML definitions for all VDL and supporting objects
- Tests and sample scripts
- Documentation
- A public network-accessible replica catalog available for use in VDS testing

VDS-1.2 does not contain:

- Globus including its RLS extensions
- Condor including Condor-G

- Java

The necessary runtime environment in terms of Globus and Condor for client side and server sides can be obtained via the Virtual Data Toolkit (VDT), see <http://www.griphyn.org/vdt/>

3.2 Release Directory Structure

There are two releases of VDS: A binary and a source distribution. The distributions differ only in the presence of the “src” directory for the source version.

The binary distribution execution environment contains several components as detailed below. Set your VDS_HOME environment variable to the top-level directory of the unpacked distribution.

Table 1: Scripts in \$VDS_HOME

setup-user-env.sh	Source this script to set-up the user environment / Bourne
setup-user-env.csh	Source this script to set-up the user environment / C-Shell

Table 2: Major directories from the base directory.

bin	contains the shell wrappers to the java classes
contrib	some unsupported contributions
com	sharable config files that frequently change (empty)
doc	documentation base directory
doc/javadoc	javadoc of all classes as far as known
doc/schemas	The generated documentation of the various XML schemas
etc	Single-machine configuration files that rarely change
example	some examples – not much there yet, refer to test/test?
lib	jar files necessary to run and/or compile
man	formatted manual pages for the CLI applications
man/man1	troff manual pages for the CLI applications
share	sharable config files that rarely change (empty)
sql	maintenance files for the SQL database backends
src	the source tree, only in the development distribution
test	an evolving battery of tests – not much there yet
var	Single-machine data files that frequently change

Table 3: Major files.

etc/properties	Java VDS property file
----------------	------------------------

etc/vdl- <i>n.nn</i> .xsd	VDLx XML schema definition, currently vdl-1.20.xsd
etc/dax- <i>n.nn</i> .xsd	DAX XML schema definition, currently dax-1.5.xsd
etc/iv- <i>n.nn</i> .xsd	XML schema definition for invocation records, currently iv-1.1.xsd
etc/poolconfig- <i>n.nn</i> .xsd	XML schema for the new pool.config file in XML, currently 1.4
etc/pool.config	The pool configuration file (to be set up by the user)
var/vds.db*	The VDLx database file, pointer, and backups
var/tc.data	The transformation catalog (to be set up by the user)
var/rc.data	The replica catalog stand-in for the shell planner only

3.3 Platform Support

The release will support most Linux systems. It was tested on various machines, running as diverse versions as Mandrake Linux 7.2, Red Hat Linux 7.1 and 7.2, and SuSE 7.3 and 8.1 with kernels 2.4.7 and 2.4.10. It is highly recommended to use a recent kernel and glibc (e.g. 2.2) on Linux. Please note that glibc at least 2.2.5 is highly recommended for developers. Regular users will have statically linked Linux files, and thus are not susceptible to some of the problems.

3.4 Requirements for the catalog and job submission site(s)

Java Release 1.4.1: from <http://java.sun.com/j2se/1.4.1/>

The following jar files are provided in the `lib` directory of the VDS distribution, for the various other packages the VDS System uses. The user environment setup script will setup the `CLASSPATH` environment variable to point to these libraries. These packages are

- Xerces 2.0.1 or 2.0.2 package for Java for processing the XML
- GNU GetOpt for Java (currently 1.0.9).
- Antlr parser generator package (currently 2.7.1).
- PostgreSQL JDBC3 connector (currently for Pg versions 7.3).
- MySQL JDBC3 connector (currently 3.0.8 for MySQL versions 4.0).

The above jar files need to be in your `CLASSPATH` environment variable. Alternatively, the jar files except for Xerces can be put in `$JAVA_HOME/jre/lib/ext` for Java 1.4. The Xerces files need to be put into `$JAVA_HOME/jre/lib/endorsed` to gain precedence over the XML libraries that ship with Java.

3.5 Run-time environment requirements

VDT 1.x installed on each submit-site; this includes Globus and Condor with the Condor-G extensions. Starting with the 1.1 release series of Chimera, the submit host Condor must be configured to be able to run *scheduler* universe jobs. In Condor speak, the submit host must run at least the *master*, *startd* and *schedd* Condor daemons. Please refer to the [Condor manual](#) for further information on setup. Condor tutorials are included for your benefit in the documentation directory. We recommend to use a “personal Condor” configuration for the submit host. A *sample configuration file* is supplied in the *examples* directory. Refer to section 4.2 for further details on the Condor configuration for the submit host.

There needs to be a grid enabled ftp (gridftp) server on each SE (Storage Element). If the submit host participates in the staging process, e.g. as source for input files, or as sink for output files, it must also feature a grid enabled ftp service.

The main means of transferring data to and from a grid enabled ftp server for this release is through the use of *globus-url-copy* (*guc*). The *guc* binary comes with the VDT Globus and any complete Globus installation. These *globus-url-copy* client needs to be installed on each of the Compute Engines (execution pools). In addition, a more sophisticated mechanism is available, named *transfer*. It works on top of *globus-url-copy* as the transfer mechanism. The *transfer* binary, provided with the VDS distribution, needs to be distributed to all participating compute sites.

All of the hosts in an execution pool must have access to a common, writable shared file system. You may want to ensure that your version of Globus features the patches from bugzilla #931, which alleviates some common problems with shared file system, gatekeeper and remote scheduler interaction.

Jobs running on compute engine can run on various schedulers through the Globus gatekeeper (e.g. jobmanager-xyz, where xyz maybe PBS, fork, condor etc). All data transfer nodes are executed in a “transfer” universe on the execution pool. This (artificial) universe is just a VDS specific universe, which is entered into the pool file. It allows to specify a different jobmanager for handling the data transfer nodes. If no entry for the transfer universe is found, the entry for the “globus” universe is picked up. A real world example would be the way that large pools are managed. All the machines in the pool may be behind a firewall for security reasons, except one machine which talks to the outside world. In this case, in order to execute the data transfers one will have to use a job manager, which will run the transfer job on the externally visible machine. The fork job manager, while being less suited for the compute jobs that need to get submitted into a local scheduling system, is ideally suited for this kind of transfer jobs.

Access to a Globus Toolkit 2.0 Replica Catalog will be optional. Users can install their own replica catalog, or use a public one provided by the VDS developers for user testing. There are two replica catalog servers installed: one at USC/ISI and the other at UofC/ANL for use by the GriPhyN and iVDGL projects. Please note that the support for the Globus 2 Replica Catalog will cease by the end of 2003.

As an alternative, the Globus Replica Location Service (RLS) is recommended to be used as a replica mechanism instead of the Globus 2.0 RC. An RLI/LRC tandem is available at UofC/ANL for use by the GriPhyN and iVDGL projects. Future releases of VDT will feature the RLS client tools.

To obtain the information needed to access either of these replica catalogs please contact chimera-support@griphyn.org.

4 Execution environment

4.1 Overview

The environment consists of the following grid elements, which are described in subsequent sections below:

- A single *catalog host* (CH)
- A single *submit host* (SH)
- One or more *compute elements* (CEs)
- One or more *compute hosts* within (ie, managed by) each compute element
- One or more *storage elements* (SEs) accessible to the compute nodes

The architecture of the VDS grid execution environment is illustrated below.

Each compute host within a compute element must have access to all files within an associated SE. Each CE is associated with one SE. An SE can be associated with multiple CEs.

A CE is assumed to be running any scheduler supported by Globus – no assumptions are made in the DAG as to the local scheduler environment. (Note that as of release Beta-4, only Condor, LSF and PBS have been tested as a compute element schedulers).

The catalog host and submit host can, but need not, all be the same host. The catalog host is the one that the catalog manipulation commands (all the commands supplied in this toolkit) execute on. The submit host is the one on which the DAGs generated by the `gencdag` command will be submitted on – i.e., the host on which `condor_submit_dag` is executed.

We assume that all executables referenced in a DAG are pre-installed at site-specific paths that are recorded in the transformation catalog (which is, in this release, a simple text file whose name is obtained from the `vds.tc.file` property, defaulting to “`$VDS_HOME/var/tc.data`”). The `gencdag` command determines the correct absolute physical pathname of an executable to use for a job in the DAG by looking up the logical executable name in the transformation catalog to determine the correct physical executable pathname for a given pool. Note that the translation is also required for executables invoked by the Pegasus concrete planner. Currently, these executables feature file staging and replica registration.

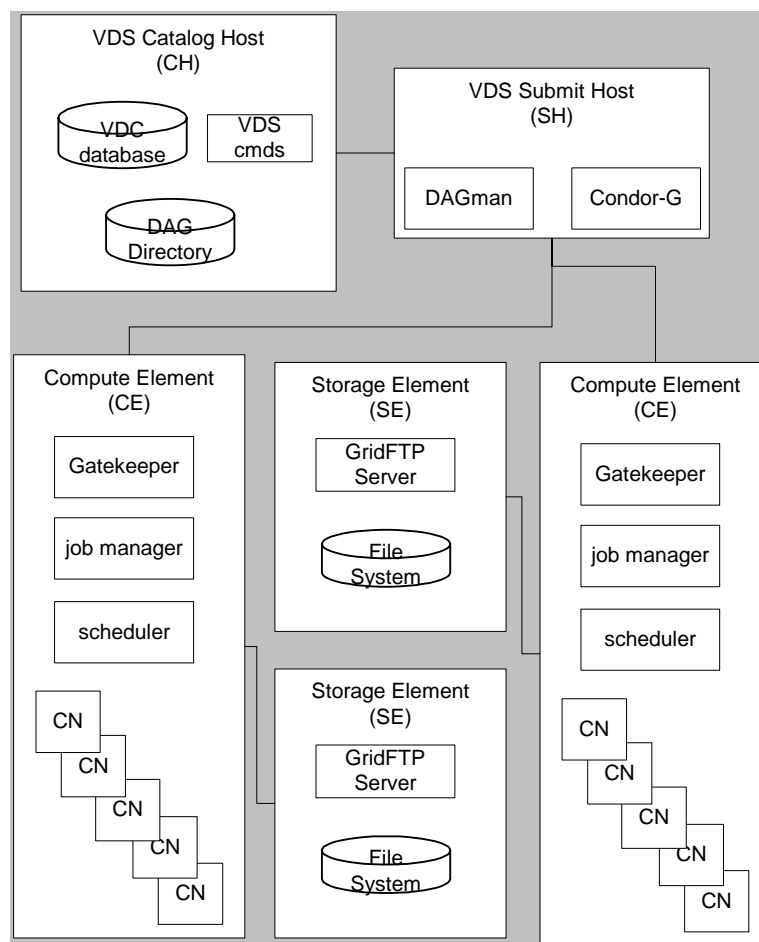


Figure 2: Grid Execution Environment for VDS

4.2 Condor setup

A submit Condor installation is required for each submit host. A complete Condor is recommended, though. The Condor installation and configuration must include the following features:

- The Condor must include to Condor-G capabilities to submit jobs to the Globus universe. If you are installing Condor yourself, you will need to add the Condor-G package after the Condor installation. Recent versions of Condor start to include the Condor-G extensions as part of Condor.
- The Condor installation must be capable of running *globus* and *scheduler* universe jobs. This requirement should be no problem for a standard or VDT-based Condor installation. Optionally, you may want to be able to run *vanilla* universe jobs.
- The Condor configuration should preferably be set up as a personal Condor. This means that non-Globus Condor jobs will *only* run on the submit host, and no other machine. The submit host *is* in effect the Condor pool.

If you install Condor and Condor-G yourself, sample configuration files *condor_config* and *condor_config.local* are provided in `$VDS_HOME/examples/condor_config`. The sample files feature some tunings for a personal Condor as described in the requirements above. The location of the configuration files depends on the Condor installation, typically `$CONDOR_CONFIG`, `/etc/condor`, `~condor`, `$VDT/etc`, `$VDT/condor_home`, `~` and `/usr/local/condor/etc`.

You should make sure to include the following configurations into the master or local configuration file, iff you configure a personal Condor:

```
start      = true
preempt    = false
suspend    = false
vacate     = false
```

These expressions should *only* be specified for a *personal* Condor. They will ensure that *vanilla* universe jobs start immediately. They will also ensure that key presses and other processes on the submit machine do not interfere with the job submission.

For a regular non-personal Condor pool, you might want to compare your configuration against the configuration included in the sample configuration file for speed-ups.



A modern Condor has a facility called the *grid monitor*. This additional process watches over *globus* universe jobs; these are remotely submitted jobs. It allows for disconnects from the remote gatekeeper, thus saving resources like gatekeeper file descriptors and gatekeeper main memory. It should be used with long-running jobs that either don't produce messages on *stdout* and *stderr*, or that can live with having their output transferred after the job is done. With short-running jobs¹, the grid monitor produces more overhead. To enable the grid monitor, add to your Condor-G configuration:

```
# Jaime's grid monitor to optimize grid submissions
GRID_MONITOR = $(SBIN)/grid_monitor.sh
ENABLE_GRID_MONITOR = TRUE
```

You might want to test your configuration first, after enable the grid monitor, as it is a very recent feature, and may not work correctly for you. Furthermore, to enable sufficient throughput of your grid jobs, you will need to configure the following Condor configuration constants:

¹ Any job of two minutes or less is *instantaneous* in grid terms.

```

GRIDMANAGER_MAX_PENDING_SUBMITS = 20
GRIDMANAGER_MAX_PENDING_REQUESTS = 200
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE = 100

```

By default, the number of jobs permissible to Condor-G is sufficient for tests, but too low for a production grid. The values above are suggestions. The limit the number of submits that may be in pending state, the number of total submits to Globus, and the number of jobs per any gatekeeper resource. Details will appear in the Condor manual.

If you are running into problems, it is always good to know exactly what is going on. Also, the developers usually require various log files in order to locate a bug or misconfiguration. For this reason, you may want to change all variables that end with the suffix `_LOG`, and change the default value of 64000 to a new value of 1024000. This will keep about 1 MB per Condor logfile, of which various exists.

Furthermore, to detect grid errors, each user usually has his or her own grid manager log file. To obtain sufficiently detailed information, set the following options:

```

GRIDMANAGER_DEBUG = D_SECONDS D_FULLDEBUG
MAX_GRIDMANAGER_LOG = 4096000

```

In certain circumstance, especially if NFS is involved, locking errors may accumulate which may lead to job failure. Please make sure that the log file in all your submit files point to a place that is not on NFS. Furthermore, if heavy NFS is involved, it may be necessary to tell Condor to ignore (spurious) NFS locking problems:

```
IGNORE_NFS_LOCK_ERRORS = True
```


If your pool is heterogeneous, and consists of machines with varying compute power, you may want to prefer to run on the powerful machines first. You can achieve this by using the following ranking in your Condor configuration:

```
RANK = TARGET.KFlops
```

[TBD: how to set up Condor's graphical job display]

4.3 Properties

Various run-time parameters to configure the VDS grid execution environment are set via a Java “properties” file. A system-wide copy is expected to be located at `$VDS_HOME/etc/properties`. The per-user properties have precedence over the system-wide properties. If present, the file `$HOME/.chimerarc` contains the per-user properties.

 Up-to-date documentation, *fresher* than the examples given below, of all properties can be found in the file `$VDS_HOME/etc/sample.properties`. The following tables show various properties, applicable at different stages in the VDS process.

<code>vds.home</code>	Allows to overwrite the <code>\$VDS_HOME</code> VDS base directory.
<code>vds.home.datadir</code>	Allows to overwrite the directory for sharable configuration files that rarely change (<i>defaults to <code>[vds.home]/share</code>, currently unused</i>).
<code>vds.home.sysconfdir</code>	Allows to overwrite the directory for machine-bound configuration that rarely change (<i>defaults to <code>[vds.home]/etc</code>, used for configuration and schemas</i>).
<code>vds.home.sharedstatedir</code>	Allows to overwrite the directory for single-machine frequently changing data

	files (defaults to [vds.home]/com, currently unused).
vds.home.localstatedir	Allows to overwrite the directory for machine-bound frequently changing data files (defaults to [vds.home]/var, used for file databases).
vds.properties	Allows to overwrite the default VDS system property location (defaults to [vds.home.sysconfdir]/properties).
vds.user.properties	Allows to override the location of the user properties file (defaults to [user.home]/.chimerarc).

Table 4: Ubiquitous VDS properties.

vds.transfer.mode	Names the transfer backend to use. If not present, the default value “single” is assumed. If present, the value “multiple” indicates the alt. transfer program.	
vds.transfer.mode.links	With the multiple transfer backend being used, this Boolean property sets whether the transfer executable should try creating symbolic links if the input files reside on the same pool as the execution pool, instead of copying them to the execution directory.	
vds.transfer.force	Boolean property which when set to true, translates to existing symbolic links being overwritten if they already exist. This is used in conjunction with vds.transfer.mode.links.	
vds.transfer.throttle.streams	The number of streams that g-u-c opens to do the data transfer.	
vds.transfer.throttle.processes	The number of g-u-c processes the transfer executable spawns while transferring multiple files between pools.	
vds.transfer.thirdparty.pools	A comma separated list of pools, where third party transfers are enabled. For these pools the transfers are scheduled to be third party instead of being the normal pull/push model. This was specifically put in for LCG sites to run their stuff.	
vds.replica.mode	Allows changing the RC backend implementation. If not present, G2RC is the default, value “rc”. A value of “rls” uses the RLI/LRC as RC backend instead.	
vds.exitcode.mode	Allows to propagate remote job failure to DAGMan, “none” does not use exitcode, “essential” ignores replica failures, and “all” runs for all jobs. (defaults to “none”)	
vds.rc.collection	rc	The Pegasus concrete planner registers all the materialized data in one logical collection which is specified by this property. By default it is GriphynData.
vds.rc.host	rc	The ldap url to the replica catalog node.
vds.rc.password	rc	The password to connect to the replica catalog.
vds.rc.manager_dn	rc	The manager DN to connect to the replica catalog.
vds.rls.url	rls	Contact string which points to a RLI.
vds.rls.query	rls	Use the more efficient “bulk” queries, which require at least a

	2.0.5 client and server software. The “multiple” mode is backward compatible. (defaults to “bulk”)
vds.dir.exec	A suffix to the Exec-mount point to determine the current working directory. If relative, the value will be appended to the working directory from the pool.config file. If absolute, it constitutes the working directory.
vds.dir.storage	Relative paths will be appended to the storage mount-point, absolute paths constitute the storage mount point.
vds.dir.create.mode	This determines the placement of the create directory nodes in the graph, that create the random directories on the execution pools where a particular workflow is executed. Value of “HourGlass” implies that the create directory nodes are placed on the top of the graph with all the other root nodes of the graph being dependant on them by placing a dummy concat job in between, creating an X shape. Value of “Tentacles” also puts the create directory nodes on the top of the graph, however they have dependencies on all the jobs that are to be run on a particular pool for which the create job is creating the random directory.
vds.db.tc	deprecated: use vds.tc.file
vds.tc.file	location of file to use as transformation catalog (defaults to [vds.home.localstatedir]/tc.data)
vds.tc.mode	Allows single or multiple reads of the TC file (defaults to “single”).
vds.db.rc	location of file to use as replica catalog for the shell planner (defaults to [vds.home.localstatedir]/rc.data)
vds.db.pool	deprecated, use vds.pool.file
vds.pool.mode	Allows “single”, “multiple” and “xml”. Refer to samples for explanations. (defaults to “single”)
vds.pool.file	location of file which has the configurations for all known pools (defaults to [vds.home.sysconfdir]/pool.config)
vds.giis.host	In XML pool-mode, this is the MDS to contact for pool information.
vds.giis.dn	In XML pool-mode, this is the distinguished name for the request.
vds.pegasus.kickstart-condor	deprecated, use vds.scheduler.condor.start
vds.scheduler.condor.start	The path to the kickstart-condor script which is used to submit to CondorG. (defaults to location \$VDS_HOME/bin/kickstart-condor)
vds.pegasus.condor-config	deprecated, use vds.scheduler.condor.config
vds.scheduler.condor.config	The path to the Condor configuration file. This value should be same as the CONDOR_CONFIG environment variable.

vds.pegasus.condor-bin	deprecated, use vds.scheduler.condor.bin
vds.scheduler.condor.bin	The path to the bin directory of Condor installation at the submit host, that contains <i>condor_submit_dag</i> script.
vds.scheduler.condor.release	Number of release instruction to retry jobs in case of transient failures. (<i>defaults to 3</i>)
vds.scheduler.condor.output.stream	Predicate, if the stdout of the remote jobs needs to be streamed back (<i>defaults to true</i>)
vds.scheduler.condor.error.stream	Predicate, if the stderr of the remote jobs needs to be streamed back (<i>defaults to true</i>)
vds.lsf.projects	obsolete; refer to vds.scheduler.remote.projects and vds.scheduler.remote.queues
vds.scheduler.remote.projects	A comma-separated list of pairs in the form pool-handle=project-name. Whenever a submit file is generated for a pool in the list, the project RSL is inserted (for accounting purposes) into the submit file. (<i>Ideally, this will become a profile entry in the TC</i>)
vds.scheduler.remote.queues	A comma-separated list of pairs in the form pool-handle=queue-name. Whenever a submit file is generated for a pool in the list, the queue RSL is inserted into the submit file. (<i>Ideally, this will become a profile entry in the TC</i>)
vds.scheduler.remote.maxwalltimes	A comma-separated list of pairs in the form pool-handle=walltime. Whenever a submit file is generated for a pool in the list, the maxwalltime RSL is inserted into the submit file.

Table 5: Properties for Pegasus.

vds.schema.dax	Points to the schema location for DAX (may be a URL) (<i>defaults to location \$VDS_HOME/etc/dax-1.5.xsd</i>)
vds.schema.vdl	Points to the schema location for the VDL (may be a URL) (<i>defaults to location \$VDS_HOME/etc/vdl-1.20.xsd</i>)
vds.schema.ivr	Points to the schema location for Invocation records (may be a URL) (<i>defaults to location \$VDS_HOME/etc/ivr-1.2.xsd</i>)
vds.db.vds	Obsolete
vds.db.*.schema	Name of a Java class that implements the database schema interface for the given catalog. Currently available are <i>ChunkSchema</i> , <i>YongsSchema</i> (<i>exp.</i>) and <i>SingleFileSchema</i> for catalog “vdc” and <i>InvocationSchema</i> for catalog “ptc”.
vds.db.*.schema.*	Depending on the database schema implementing class, it may require further properties. These are kept in the vds.db.*.schema property namespace.
vds.db.driver	Name of a Java class that implements the database driver interface. Currently available are <i>Postgres</i> and <i>MySQL</i> . Further drivers are being prepared
vds.db.driver.url	Database name to use as access path, e.g. jdbc:postgresql: plus the user’s account name (Java property [user.name]).

vds.db.driver.user	Username for the database, defaults to the user's account name (Java property [user.name]).
vds.db.driver.password	Password for database access, defaults to the user's account name (Java property [user.name]).
vds.db.driver.*	Depending on the database implementing class, further properties may be required. These are kept in the vds.db.driver namespace, and will be copied to the JDBC connect call properties by removing the VDS namespace prefix.
vds.log.*	Allows to output debug messages for various modules (instead of asterisk). The value is the filename, or one of the special names stdout or stderr.

Table 6: Properties for VDC manipulations and abstract planning.

vds.replica.mode	rls
vds.rls.url	rls://sheveled.mcs.anl.gov
vds.exitcode.mode	all
vds.transfer.mode	single
vds.db.vdc.schema	ChunkSchema
vds.db.ptc.schema	InvocationSchema
vds.db.driver	Postgres
vds.db.driver.url	jdbc:postgresql:\${user.name}
vds.db.driver.user	\${user.name}
vds.db.driver.password	\${user.name}

Figure 3: Example property file

If the vds.pool.file and vds.tc.file properties are not specified, the planner searches in \$VDS_HOME/etc for a file named pool.config and in \$VDS_HOME/var for a file tc.data respectively.

If the corresponding properties *vds.schema.vdl* and *vds.schema.dax* are not set, the schema files are searched for in their default location \$VDS_HOME/etc/<schema.xsd>. The XML root element attribute xsi:schemaLocation="URI URL" in any XML document is only a hint. The existence of a locally stored schema with a user-selectable position eliminates the need to be online for the XML parsers to work. Also note that the latest version of our classes and examples default document schema URLs to their web location.

4.4 Virtual Data Catalog execution environment

The VDC currently has different storage implementations that can be chosen by a user. The VDC can work with a single XML file that stores all virtual data definitions. This approach is reasonably fast for up to 1000 definitions. Being the simplest approach, if you want to try out Chimera, it is the default, if you don't set up anything.

If you require working with more than 10000 definitions, a real database is highly recommended. Currently, Chimera supports PostGreSQL and MySQL as valid SQL databases. More database backends are expected to be supported later. Other options, e.g. directory based hashed storage, may be investigated as time permits.

The database schema is a separate layer on top of the database implementation. Figure 4 shows the layering of the database access through VDC. On the top VDC commands perform standard operations on data. The

VDC layer in turn accesses the database schema layer to fulfill its function. The database schema layer implements to a given SQL schema (or skips databases all together, as in the single file case), using the database driver layer to perform the actions. Most of the operations in the driver layer are hand-throughs to the JDBC3 layer to actually perform any action on the remote database.

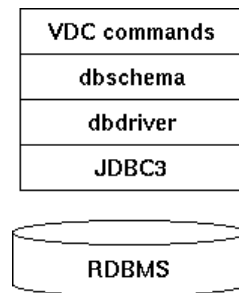


Figure 4: Layering of database access.

The layering approach was chosen for two reasons:

1. The separation between database schema and database driver allows to focus within each part on its strengths.
2. The database driver on top of JDBC was necessary to mediate between JDBC3 optional features, work-around implementations for specific databases, and general non-conformity of DMLs.

The rest of this section deals with the available database schemas first, and then with the database driver layer. In the latter, the database-specific configuration will be explained.

4.4.1 The VDC database schema layer

The database schema layer defines a simple API to access a minimalist subset of virtual data to perform the basic operations insertion and update, deletions, and searches and routing. Any class deriving from the `DatabaseSchema` class is permitted as legal schema implementation. If a user provides her own schema implementation, she needs to point her `CLASSPATH` to her class, and add the fully-qualified Java class name to the `vds.db.schema` property.

4.4.1.1 The single file based default VDC

The virtual data catalog resides in a file (in a designated “catalog” directory) which contains all active VDC definitions. All commands that process the catalog will read the entire VDC XML file into memory before processing anything, and all files that update the catalog will write the VDC XML file back when done.

The single-file approach is the default database schema, also known as *SingleFileSchema* default value for the `vds.db.schema` property. The catalog directory also holds, for backup purposes, a set of files that contain previous revisions of the current file. The name of the directory and file to use are communicated to the Java code and command-line applications through the command line option “-d dbprefix”². A default will be picked from the property “`vds.db.schema.file.store`”, which in turn defaults to `$VDS_HOME/var/vds.db`.

Each VDS command will operate on one VDC per command invocation. All VDS commands must execute on a host that can access and lock files in the catalog directory. We refer to this host as the “catalog host”. Due to Linux NFS file locking issues, we recommend that the database files be placed on a locally mounted filesystem.

² At the moment, the “-d dbprefix” feature in Chimera is temporarily deactivated.

Users can maintain any number of VDCs, but only one catalog-modifying command at a time can operate on a given VDC. (An “updating command” is one that makes changes to the VDC, i.e., insertvdc, updatevdc, deletevdc). Attempts to run multiple updating commands on a single VDC concurrently would result in all but the first command being blocked, through UNIX file locking operations.

The -d option (on VDS commands that access the VDC) specifies the directory and the filename prefix (i.e., the “basename”) of the database file to use. We will refer to the value of the -d option as \$dbprefix in the description of this mechanism. The database files that will be used (in a rotating fashion) are \$dbprefix.0 through \$dbprefix.9. Each invocation of a command that updates the VDC will read the file indicated by the indirect pointer file \$dbprefix.nr, and produce the next file in the circular cycle \$dbprefix.0 through \$dbprefix.9. \$dbprefix.nr will always contain a single ASCII digit 0 to 9.

If \$dbprefix names an existing file, it indicates the name of an initial database to use. The output database in this case will be \$dbprefix.0. The initial database file will only be used once, and will not thereafter take part in the cycle of input or output files.

The described file rotation approach applies similarly to property-chosen file locations for the VDC.

If \$dbprefix is not a file, \$dbprefix.nr will be read to determine the current file in the cycle. If any inconsistencies are detected among the files by a VDS command that opens the catalog, that command will halt without making any updates. If a VDC command finds that any files in the cycle \$dbprefix.[0-9] exist, and no .nr file is found, the command will silently create an initial database. Missing output files in the cycle are silently created; existing output files in the cycle are silently overwritten.

4.4.1.2 The directory-based multi-file VDC

The directory-based VDC was an experimental feature. Since it did not prove to be as efficient as envisioned, it was removed from the VDS.

4.4.1.3 The fully structured SQL-based VDC

The fully-structured fine-grained VDC has been temporarily removed, as its performance was not sufficient. For metadata research, though, it will be re-introduced.

4.4.1.4 The minimal structured SQL-based VDC

The chunk backend is a minimally structured approach which uses the database as fast and large storage for XML definitions. Currently, the chunk backend expects to work with PostgreSQL. More support for different database engines will be forthcoming.

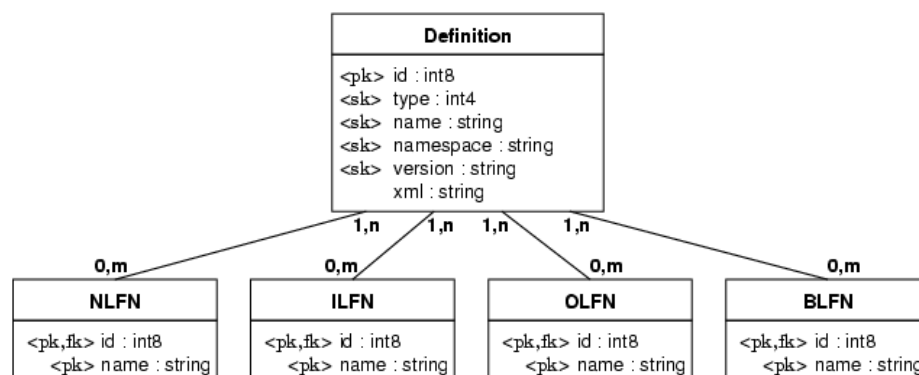


Figure 5: The database schema for minimal structure.

Figure 5 shows the schema for the minimally structured database. The diagram is in UML-like fashion with the following deviations. The diagram shows sequences as primary keys, as denoted by a <pk>. The

secondary keys <sk> are inserted for documentation purposes. <fk> denotes foreign keys, although the foreign key constraint is currently being avoided for speed reasons. The dashed line denotes a logical connection that is not modeled nor enforced in the database.

Only the secondary key of any definition, that is the fully-qualified definition name (the triple of namespace, name and version) plus the kind of definition (either a transformation or a derivation) is exposed through the secondary key. Every other detail is hidden in the XML-stored string within the database. In order to route provenance efficiently, the LFN of each definition is externalized. For performance reasons, each linkage (none, input, output, both) is kept in a separate table.

In order to use the “chunk” version, you need to set the property `vds.db.schema` to “ChunkSchema”. Please refer to the next sections on how to set-up a specific database backend driver. You will need to run the appropriate SQL script to create a schema. All SQL scripts start out with either “create” or “destroy”, followed by the name of the schema, followed by the name of the driver.

4.4.2 The Provenance Tracking Catalog (PTC) schema

The provenance tracking catalog is still an experimental feature, and the database schema may change in the near future to accommodate better provenance at less cost. Figure 6 shows the current database schema for the provenance tracking.

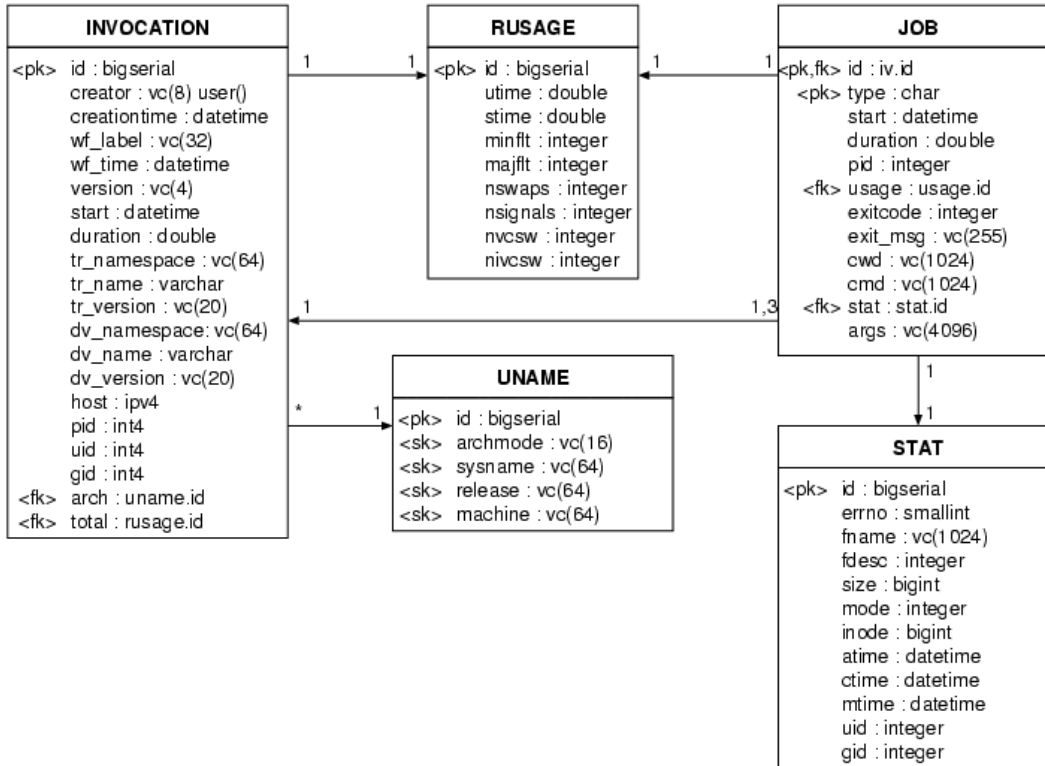


Figure 6: The Provenance Tracking Catalog's schema.

4.4.3 VDC and PTC combined database drivers

The database drivers perform the transaction with the database itself. The driver also provides additional information about the capabilities of the underlying JDBC3 driver to the schema, so that the schema may chose alternatives accordingly.

Similar to the schemas, database drivers are dynamically loaded. Any database driver that derives from the `DatabaseDriver` class is permissible. If a user wants to provide her own database driver

implementation, all she needs to do is point her CLASSPATH to her class, and provide the fully-qualified Java class name to the vds.db.driver property.

4.4.4 PostGreSQL 7.3

4.4.4.1 A user-owned installation

The Postgres database allows for a user installation that can be shared – it is not necessary to have a system installation, although that is the preferred installation. In order to obtain a user installed Postgres, the following steps are required:

1. Download the latest 7.3.* version by following the download link from <http://www.postgresql.org/>. You will first need to select a mirror site, and then follow into the source directory.
2. Unpack the tarball at a place of your convenience. This will be the build directory. You also need to decide where to install the runtime environment.

```
gzip -cd postgresql-7.3.4.tar.gz | tar -xf
```

3. Change into the unpacked distribution

```
cd postgresql-7.3.4
```

4. You need to configure postgres for your platform. Recent versions of gcc with recent versions of Postgres will not allow for maximum optimization without breaking triggers. The following optimizations were found to be safe – you will need to determine your own safety factor by running the self-test. The following is an example:

```
JAVA_HOME=/usr/lib/java JAVACMD=$JAVA_HOME/bin/java \
CC=gcc CXX=g++ \
CFLAGS='-pipe -O -funroll-loops -mcpu=i686 -march=i686' \
./configure --prefix=$HOME/pgsql \
--disable-nls --with-pam --enable-integer-datetimes \
--with-java --with-perl --with-tcl --with-cxx \
--with-openssl --with-pgport=12345
```

The first line sets the path to your Java installation and the Java binary. Please note that you will only need to set this, if you enable `--with-java` as language binding. Enabling Java mandates that you **must** have ant³ installed and its binaries visible in your PATH.

The second line configures the compilers to use. You may skip this line.

The third line sets the optimizations specific for a GNU C compiler. Do not use higher optimization, or you will break things in the database.

The configure script starts out by choosing the installation location. Shown is an installation in the users HOME directory. If the HOME is NFS-mounted, you should prefer a local disk instead!

The fifth line disables native language support. It is an option that is on by default, and makes things usually only slower, not better. It enables the privacy authentication module infrastructure (available on Linux and Solaris) to potentially log-in a user. You may chose to not provide this option. The final option enables the use of 64bit integers to record SQL timestamps. This is usually well for modern Linux systems.

³ See: <http://ant.apache.org/>

The sixth line deals with language inclusions. You should at least include the C++ language binding, in order to enable the language binding configuration within postgres. You may chose to drop any of the Java, Perl or TCL language binding. For Java, you must have ant installed. For Perl, your perl must be compiled to use a libperl, which most perls do not. For TCL, your TCL configuration must be available at configuration time.

The final configuration options allow for the usage of SSL to encrypt your connections. This features is optional. Also, you can specify the default port of the database engine to listen at. For a user installation, you should set this port to something other than 5432, the Postgres default port. For a root installation, you should not set the port.

If the configure script dies on any error, please remove offending options as necessary and re-run.

5. Compile Postgres. You will need a recentish version of GNU make for this:

```
make
```

If the compilation died on any error, first clean up everything

```
make distclean
```

and then return to step 4. Remove any offending pieces from the configuration.

6. Check the integrity of the compiled system. This step is very important, and should not be skipped. Also, this step must be executed as non-root user:

```
make test
```

If any of the tests fail despite your previous attempts, chances are that the optimization level is still too high. Among the first things to fail are triggers. If any of the tests fail, clean out everything

```
make distclean
```

and then return to step 4. Remove any offending pieces from the configuration.

7. If the tests succeeded, you can install postgres. If you install as non-root user, Postgres will assume a user installation, and the installing user becomes the database super-user. If you do a root installation, you must have a user “postgres” in your system. For root installation, please follow the instructions in the Postgres manual.

```
make install
```

8. Please set up your PATH to include \$HOME/bin and your LD_LIBRARY_PATH to include the \$HOME/lib directory. You may want to put this into a login script for the account.

9. Initialize the database. Assuming a user installation, you simply need to run

```
initdb --pgdata=/path/to/data/directory
```

The path to the data directory is the place where your postgres installation will write its database files. This must be a local disk. This must *never* be any NFS storage. International user may run into trouble, if certain locale settings are visible during the initialization of the database, but not during the use.

The above initialization will take only a few seconds. It will create the data directory, configuration files and initial database files.

10. Before you can use a user installation of postgres, you must enable certain options. You may want to check with the Postgres user manual for higher optimizations (e.g. System-V shared memory), and how to configure your system. For starters, you may want to edit the \$PGDATA/postgresql.conf file to include the following options:

```

tcpi_socket = true           # permit internet logins
hostname_lookup = false      # speed up connects
show_source_port = true     # tracing
port = 12345                 # verify that it matches

shared_buffers = 256         # min 16, typically 8KB each
max_fsm_relations = 4096     # min 10, ~40 bytes each
max_fsm_pages = 131072      # min 1000, ~6 bytes each
max_locks_per_transaction = 64 # min 10
wal_buffers = 32             # min 4, typically 8KB each

sort_mem = 32768             # min 64, size in KB
vacuum_mem = 32768           # min 1024, size in KB

fsync = false                # slightly unsafe, a lot faster
log_connections = true

```

Also, before you can connect from the internet to your database, you must include some form of permissions to the permission table. Postgres only allows “localhost” to connect via the internet by default. If you need more machines, including your primary interface, you need to edit the file `$PGDATA/pg_hba.conf`. The file contains plenty of comments to show-case different options. It is recommended to encrypt the password exchange of remote logins with either “crypt” (old method) or “md5” (new method).

#	TYPE	DATABASE	USER	IP-ADDRESS	IP-MASK	METHOD
local	all	all				trust
host	all	all		127.0.0.0	255.0.0.0	trust
host	all	all		128.135.11.0	255.255.255.0	md5
host	all	all		140.221.8.0	255.255.248.0	md5
host	all	all		192.168.0.0	255.255.0.0	md5

11. You may now start your user-installed version of Postgres. It is recommended to use the Postgres controller program, e.g.:

```
pg_ctl -D $PGDATA -l /where/to/log start
```

Please be aware that each Postgres installation eats a chunk of your memory and other resources. Postgres installs nicely as a system database, too, and that is the preferred mode of installation and operation.

4.4.4.2 PostGreSQL 7.3 setup for the VDC

The following steps are required to set-up the PostGreSQL driver and backend:

1. The administrative user “postgres” (or the database super-user in case of a user install) needs to create the database user for you, usually the sequence, where `<username>` is a placeholder for the real account name:

```

psql -U <dbsuperuser> template1
create user <username> password '<username>' createdb;
\c - <username>
create database <username>;

```

If your database engine runs on the same host, you will connect through a Unix socket. If your database engine resides on a different host, you may need to specify the `-h` host and `-p` port for the database engine. In case of multiple user installations, you may also prefer to name the database host and port specifically to avoid confusion which engine is serving your request. Note that the `pg_hba.conf` file must be set up to allow internet access, and the engine must be configured to permit internet sockets.

2. For each database, the user has to activate the PL/SQL language once and only once per database with the following command:

```
createlang plpgsql <username>
```

3. Once the account is active, the schema needs to be created. If you use non-default settings that differ from the ones described above, please refer to the manual page for *psql* for details on how to use different databases and usernames. Usually, creating the schemas is as simple as running the *create-postgres.sql* file through *psql*, e.g.:

```
cd $VDS_HOME/sql
psql -f create-pg.sql
```

In case of multiple user installations, you may want to specify the engine's host and port, the database name and the database username under which to construct the new database in order to avoid confusion.

4. Make sure that the PostgreSQL JDBC driver is accessible in your CLASSPATH. A version *pg73jdbc3.jar* is provided for you convenience in the *\$VDS_HOME/lib* directory.
5. Finally, the properties need to be set accordingly.
 - a. Point `vds.db.driver` to the value "Postgres" (without the quotes).
 - b. With JDBC, a database is represented by a URL (Uniform Resource Locator). The `vds.db.driver.url` property reflects this URL. If PostgreSQL runs locally, you can omit the host portion. If your database server runs remotely, please make sure that you can access it. With PostgreSQL, the URL to access any database takes one of the following forms:
 - `jdbc:postgresql:database`
 - `jdbc:postgresql://host/database`
 - `jdbc:postgresql://host:port/database`

where:

<i>host</i>	The host name of the server. Defaults to localhost.
<i>port</i>	The port number the server is listening on. Defaults to the PostgreSQL standard port number (5432).
<i>database</i>	The database name.

- c. You must set the username and password for the database access, typically your UNIX account name. Please set the `vds.db.driver.user` and `vds.db.driver.password` properties for PostgreSQL.

You are now ready to the PostgreSQL based VDC.

4.4.5 MySQL 4.0

4.4.5.1 MySQL 4.0.* user-owned installation

Unfortunately, we cannot help you with this.

4.4.5.2 MySQL 4.0 setup for the VDC

The following steps are required to set-up the MySQL backend:

1. The administrative user needs to create your account and the database for your account. Please refer to the MySQL documentation on how to accomplish this. It is not trivial, and we cannot help you!
2. Once the account is active, the schema needs to be created. If you use non-default settings that differ from the ones described in the properties section, please refer to the manual page for *mysql* on details, e.g. how to use different databases and usernames. Usually, creating the schemas is as simple as running the *create-my.sql* script through *mysql*, e.g.:

```
cd $VDS_HOME/sql
mysql -u username -p databasename < create-my.sql
```

Please substitute *databasename* above with the name you gave your database. The default is the same name as your UNIX account. Security usually requires you to provide a typed password, and your username.

3. The administrative user might need to reset the privileges in table *user* in database *mysql* to make the database accessible to you.
4. Make sure that the MySQL JDBC driver is accessible, and in your CLASSPATH. A version of the driver *mysql-connector-java-3.0.8-stable-bin.jar* is available in the *\$VDS_HOME/lib* directory.
5. Finally, the properties need to be set accordingly:
 - a. Point `vds.db.driver` to the value "MySQL" (without the quotes).
 - b. With JDBC, a database is represented by a URL (Uniform Resource Locator). The `vds.db.driver.url` property reflects this URL. If MySQL runs locally, you do not need to set the host portion, although JDBC always accesses MySQL through its networked interface. If your database server runs remotely, please make sure that you can access it. With MySQL, the URL to access any database takes one of the following forms:

- `jdbc:mysql:///database`
- `jdbc:mysql://host/database`
- `jdbc:mysql://host:port/database`

where:

<i>host</i>	The host name of the server. Defaults to localhost.
<i>port</i>	The port number the server is listening on. Defaults to the MySQL standard port number (3306).
<i>database</i>	The database name.

- c. You must set the username and password for the database access, typically your UNIX account name. Please set the `vds.db.driver.user` and `vds.db.driver.password` properties for MySQL.
- d. Please refer to the sample properties for further properties that are understood by the MySQL driver. Please note that these properties will reside in the `vds.db.driver` namespace.

You are now ready to the MySQL based VDC.

There is a known bug with extremely huge VDL statements to be stored in MySQL. In the case that you use many arguments, and your insertion or update fails for mysterious reasons, you might want to consider setting the following parameter in your *my.cnf* file:

```
[mysqld]
set-variable = max_allowed_packet=20M
```


The above statement will increase the packet size that can be passed via the network from the default 1MB to 20MB instead.

4.4.6 Other database engines

A number of other database management systems are being researched, but those are not available yet.

4.4.6.1 SQL Server 2000

This database driver is not yet available.

4.4.6.2 Oracle 8i

This database driver is not yet available.

4.5 DAG Execution - Grid Environment

DAG nodes are submitted to Globus GRAM gatekeepers of various remote sites employing Condor-G on the submit host. The submit files for Condor are generated by the *gencdag* command.

Depending on the option (`--submit`) supplied to the *gencdag* command, Pegasus will either submit the generated DAG to the local Condor-G, or will generate a DAG to be submitted manually using the Condor command *condor_submit_dag*. The latter is the recommended mode of operation.

The replica registration jobs are scheduled for execution on the submit host, and thus are run in the "scheduler" universe of the submit host. All remaining job submit files, staging and application, have the specification "universe = globus", which indicates remote execution. Appropriate RSL strings are generated to indicate whether the job has to be run in the vanilla or standard universe on a remote Condor pool, and to set the remote working directory. For other scheduling systems, e.g. PBS, the vanilla universe should be specified in the pool.config entry for the site.

The `--p` option selects one or more remote pools on which to execute computations. Application jobs are only executed on those pools that actually contain a mapping from the logical transformation for the given universe in the TC (tc.data file). A pool from multiple candidates is randomly picked. The job manager to which the jobs are submitted to are picked up from the pool.config file. The jobmanager to be used for DAG execution is determined by the key-pair "pool, universe", where universe is the Condor universe which the job will be executed in. For non-Condor schedulers, the universe is always "vanilla".

The pool.config file is expected to be found in `$VDS_HOME/etc` directory unless overridden by the `vds.db.pool` property. This pool.config file contains the following white-space-separated fields:

1	Pool	Any valid location registered in the replica catalog or as a pool attribute in the Replica Location Service (more specifically in the local replica catalogs) providing computational and/or storage facilities. The special pool named "local" (without the quotes) denotes the storage/compute element that reflects submit host and not a remote pool.
2	universe	The second column contains a universe that is supported by the pool. Some restrictions apply:

		vanilla	job manager string should <i>not</i> point to a fork job manager
		standard	job manager string must point to a Condor job manger
		globus	this fork-jobmanager pointer is deprecated.
		transfer	job manger string can point to any jobmanager, depending on your firewall settings. The fork-jobmanager is recommended.
3	jobmanager contact string	The jobmanager contact string allows to access different jobmanagers for different purposes. Refer to the pool column above for restrictions and recommendations. For the "local" pool, the absence of a jobmanager contact is specified by using "null" (without the quotes).	
4	url-prefix	Depending upon which replica mode is used, this entry has different semantics. It is "null" (without the quotes) in case of Replica Catalog, as URL Prefixes are picked up from uc attribute of the location objects. In case of the Replica Location Service this string refers to (protocol + "://" + hostname + se-mount-point) e.g. gsiftp://birdie.isi.edu/nfs/asd2/vahi/Pegasus/GriphynData. /nfs/asd2/vahi/Pegasus/GriphynData would be the se-mount-point for the pool.	
5	Workdir/exec-mnt-point	This entry points to a shared file space on the remote pool where execution commenced, and where files are maintained. See also: [vds.dir.exec]	
6	gridstart	This entry effectively points to the remote gridstart installation. Although execution through gridstart is recommended, a value of "null" (without the quotes) indicates that execution through gridstart is not wanted.	
7	LRC pointer	If RLS is used as replica catalog, this field records the LRC responsible for files kept on the named pool. Only the "vanilla" universe requires this entry. Otherwise, use the value of "null" (without the quotes).	

Table 7: Columns in the pool.config file

All data transfer jobs in the DAG are executed in a *transfer* universe on the execution pool. This universe is a Pegasus-defined artificial universe which is entered into the pool file. Its purpose is to designate a specific jobmanager for handling the data transfer nodes. If no *transfer* universe is found for a given pool, the entry for the *globus* universe for a given pool is used to perform the data transfer.

The registration of the materialized data is accomplished by Pegasus by inserting replica registration nodes into the workflow of the concrete DAG. For the G2RC the script "replica-catalog" is executed. It is located in the bin directory of the release. The script is referred to by the logical name of *GriphynRC* from the transformation catalog (tc.data file). Thus the transformation catalog must be populated with a record with fields *pool = local*, and *logical name = GriphynRC* which points to the VDS installation on the submit host.

For the Replica Location Service the script "rls-client" is executed. It is located in the bin directory of the release. The script is referred to by the logical name of "*RLS_Client*" from the transformation catalog. Thus the transformation catalog must be populated with a record with the fields *pool = local*, and *logical name = RLS_Client* which points to the installation on the submit host.

The pool named *local* designates the submit host where Condor-G is running. It is recommended to be different from the execution pool. Since the replica registration jobs run the replica-catalog commands distributed with VDS, the submit host needs a Java and VDS installation identical to the catalog host (CH).

Each pool must contain an entry corresponding to *universe = vanilla* to designate the entry from which the concrete planner determines the execution mount point for the pool. This is used to determine the working directory in which the jobs are executed at the remote pool.

When using the Pegasus-supplied *replica-catalog* command to populate the replica catalog, the ‘re’ attribute for the location object is set when creating a collection. More details on this are found in the man page for the Pegasus replica-catalog command.

To execute a *gencdag*-generated DAG locally (on the submit host), specify the execution pool argument to *gencdag* as “local” and populate the pool file accordingly. The “local” pool is a reserved pool handle. Any jobs in the “local” will be executed on the submit host itself, either in its scheduler or standard universe. It avoids passing through a gatekeeper and thus Globus, by directly submitting either to Condor. For the “local” pool handle to work as described, a personalized Condor must be configured, see section 3.5. One example for local execution would be to run the black-diamond (a test case provided with VDS as test2) on localhost, and asking for the materialized files at location "isi".

Below are examples for a pool.config and a tc.data file. Further examples can be found in `$VDS_HOME/test/pool.config` and TC files in subdirectories thereof.

#Pool	Universe	Job-mgr-String	url-prefix	Exec-mnt-point	gridstart	LRC-url
local	vanilla	null	null	Null	/bin/kickstart	null
local	transfer	null	null	Null	/bin/kickstart	null
isi	transfer	birdie.isi.edu/jobmanager	gsiftp://smarty.isi.edu/home/gmehta	/nfs/asd2/gmehta	/bin/kickstart	rls://mantle.isi.edu
isi	vanilla	birdie.isi.edu/jobmanager-condor	gsiftp://smarty.isi.edu/home/gmehta	/nfs/asd2/gmehta	/bin/kickstart	null
isi	standard	birdie.isi.edu/jobmanager-condor	gsiftp://smarty.isi.edu/home/gmehta	/nfs/asd2/gmehta	/bin/kickstart	null

Figure 7: Example pool.config file

#Pool	Logical transformation	Physical transformation	Environment Variables
local	GriphynRC	/vds-1.0b4/bin/replica-catalog	JAVA_HOME=/smarty/jdk1.31._03;VDS_HOME=/vds-1.0b4
isi	findrange	/smarty/testdays/keg	Null
isi	preprocess	/smarty/testdays/keg	Null
isi	analyze	/smarty/testdays/keg	Null
isi	globus-url-copy	/vdt/bin/globus-url-copy	GLOBUS_LOCATION=/vdt; LD_LIBRARY_PATH=/vdt/lib

Figure 8: Example tc.data file

4.6 Profiles

A VDS transformation (TR) may contain multiple profile instances from four possible *profile groups*, listed below. Each profile instance is a name-value pair that controls some aspect of the execution of the transformation within a derivation. Profiles can specify numerous job attributes, define below. The abstract planner copies the profile information from the VDC into the DAX file where the concrete planner can use them to determine several characteristics of the concrete DAG.

The profile groups currently defined are:

1. **condor** profile instances contains settings that go into a Condor submit file, e.g. *universe=*, *requirements=*, etc. Condor *name=value* pairs go into the condor submit file of every DAG node generated for this TR. One should avoid using options that are regularly used in the submit file, or that may clash with other profiles or the operation of Pegasus jobs, i.e., the Condor-G “globusrl”,

“globusscheduler” or “environment” keys. Doing so will yield either unexpected results⁴ or non runnable jobs in the DAG.

2. **dagman** profile instances may contain DAGMan specific parameters like retries, pre- and post-job scripts. The name=value pairs go into the DAGMan .dag file and pertain to all jobs constructed for this TR. Please note that in the current release, the *dagman* profile type is not implemented.
3. **env** profile instances specify environment variables. Environment name=value pairs go into the UNIX environment for any DAG node of this TR. Do not use a Condor.environment nor a Globus RSL string to set the environment. Additional environment variables from the transformation catalog will be merged, with TC environment variables having precedence.
4. **hints** profile instances specify execution and planning parameters like “exec-lfn” which designate the logical filename of an executable and “exec-universe” which specifies the execution-node universe a job should run in. Keys starting with “exec” are proposed future names, and not implemented in this release. Currently, only few keys live in the hints profile space:
 - a. **pfnHint**: The *pfnHint* option allows to override an mapping from the transformation to an application in the transformation catalog. The value is the complete path name to the application. For this reason alone, the use of this option is not recommended.
 - b. **pfnUniverse**: This is the universe to run jobs for a given transformation in. You will only need this key, if you plan to run some jobs in the Condor “standard” universe. If not specified, the universe defaults to *vanilla*, which means the jobs derived from this TR can run under any scheduler at the pool.
 - c. **globusScheduler**: This option allows to specify an alternative scheduler to use instead of the one chosen by the concrete planner.

Profiles in VDL are only specifiable on TRs, and cannot currently be overridden by DVs. Profiles can be specified using variables in a TR, and the DV can pass parameters to the TR, which in turn modify the outcome of the profile. Profiles remain associated with the TR in the DAX for use at run time, and pertain to every DV of the TR.

4.7 Security

The VDS commands that run on the Catalog/Submit node will run under some specific user-level UNIX id, as well as under a corresponding GSI identity, obtained by issuing `grid-proxy-init` before submitting the DAG to Condor-G.

Access to the VDC database is controlled only by the rules of UNIX file security in this release. Some databases allow encryption, which is outside GSI.

4.8 File naming, Data Transfer, and Replication

This section describes the handling and translation of file names from VDL statements all the way through to runnable DAGs that `gencdag` generates for execution on a grid. It details the process by which names are represented in the virtual data catalog, and then translated and made accessible to running jobs. Examples at the end of this section attempt to clarify the somewhat detailed explanation.

Each execution site consists of a cluster, all of the nodes of which must have shared access to a common file system. Stated in Grid terms, each compute element has an associated storage element that is directly

⁴ For the interested user: Condor does not merge multiple entries with the same key in its submit files. Thus, the last entry in the submit file will “win” out over previous ones. The “win” situation might yield submit files that are rejected by Globus. Using a Condor environment option will win over the merged variables from the env profile and TC.

accessible to the compute nodes CE, and the shared file system(s) of the SE are mounted below an identical mount point on all CNs of the CE.

The logical names in the abstract dag refer to the names of data files or the names of transformation executables. The logical to physical mapping for the data files is done using the Replica Catalog or the Replica Location Service depending on the replica mode specified, while for transformations the mapping is done via a Transformation Catalog (currently implemented in the flat file *tc.data*).

The following description introduces various filename concepts. Later, we will show limitations of the actual implementations. We distinguish between the following types of file names and file name segments:

- *Logical filenames* (LFNs) are used in VDL “DV” statements, as either “in” or “out” parameters, or as executable names. LFNs should only be relative (do not start with a “/”).
- *Physical filenames* (PFNs) are those referenced by code at run time, on command lines, in parameter files and in functions such as `open()`. All physical filenames generated by *gencdag* are absolute (i.e., start with a “/”).
- *Storage file names* (SFNs) indicate the names of files on the SEs of compute elements. The replica catalog maps LFNs to (pool, SFN) pairs. (Note that the replica catalog “location” denotes a “pool”).

Note that when using the Globus2 Replica Catalog, the LFN and SFN are always the same – the replica catalog simply determines which locations the LFN exists at, but assumes that the physical copies of the logical file will be named the same at all locations.

In case of the Replica Location Service, the SFN and the LFN also need to be same, but the PFNs can point to different directories on the same pool. Please note that this behavior is not possible if using the Globus2 Replica Catalog.

For example, suppose two LFNs “foo” and “foo1” exist in pool “isi”. If the Globus2 Replica Catalog is being used, a location object with following attributes needs to be set up:

```
re = isi
uc = gsiftp://birdie.isi.edu/nfs/asd/Pegasus/GriphynData
```

In this case, the following mappings are stored in the G2RC:

```
lfn = foo    pfn = gsiftp://birdie.isi.edu/nfs/asd/Pegasus/GriphynData/foo
lfn = foo1   pfn = gsiftp://birdie.isi.edu/nfs/asd/Pegasus/GriphynData/foo1
```

If using the Replica Location Service, the (lfn,pfn,pool) tuple needs to be registered with the LRC corresponding to the pool “isi”. Registration is best done using the “*rls-client*” tool, which comes with this distribution. In the case of RLS, following mappings are possible (note the directories):

```
lfn = foo    pfn = gsiftp://smarty.isi.edu/nfs/asd2/gmehta/GriphynData/foo
lfn = foo1   pfn = gsiftp://smarty.isi.edu/nfs/asd2/vahi/Pegasus/foo1.
```

- *Transfer file names* (TFNs) are the full URLs that will be used by utilities such as *globus-url-copy* to reference physical file names. In case of the Replica Catalog TFNs are formed by prefixing the URL-prefix (the uc attribute of the location object) to the SFN, whereas in case of Replica Location Service the TFNs are formed by taking the PFN entries in the LRCs corresponding to the respective pools.
- A *URL prefix* is the leading portion of the URL that is prepended to the PFN. This is the uc attribute of the location object in the Replica Catalog or the url-prefix entry from the pool.config file if one is using the Replica Location Service.
- An *SE prefix* (also called the *SE mount point*) is an absolute pathname, typically of the mounted root directory of a storage element. It is prepended to the front of the SFN obtained from the replica

catalog to form the PFN. The SE prefix comes from the replica catalog. In case of the Replica Location Service, the se-prefix is derived from the url-prefix entry in the pool.config file.

- Note that the terms pool, location, and site, all refer to the same entity.

A replica catalog interface maps LFNs to zero or more (pool, SFN) pairs. The (pool, SFN) pair in turn maps the SFN into both a TFN and a PFN. Files are transferred using the TFN and physically accessed using the PFN. This process is illustrated in Figure below and summarized by these rules:

$SFN = LFN$ (if using GT2 Replica catalog)

$TFN = url\text{-}prefix/SFN$

$PFN = se\text{-}prefix/SFN$

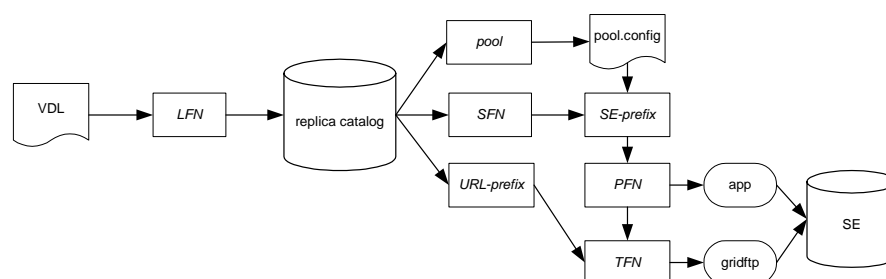


Figure 6: File name translation

Some examples of the filename translation processing are presented here to clarify this process. The examples shown here are based on the Globus 2.0 replica catalog, hence the SFN of each file is the same as its LFN. Assume three locations, loc1.edu, loc2.gov, loc3.org, and that the replica catalog has the following contents:

```

Catalog myrc, Collection c1
  Filenames: f1, f2, f4,          Location loc1.edu
    URL prefix = gsiftp://loc1.edu/raid1/prod
    Filenames = f1, f2
  Location loc2.gov
    URL prefix = gsiftp://loc2.gov/data/stg5
    Filenames = f1, f4
  Location loc3.org
    URL prefix = gsiftp://loc3.org/ul/workdata
    Filenames = f2, f4
  
```

To access f2 at loc1.edu, SFN=f2, PFN=/raid1/prod/f2, TFN=gsiftp://loc1.edu/raid1/prod/f2

To access f4 at loc2.gov, SFN=f4, PFN=f4, TFN=gsiftp://loc2.gov/data/stg5/f4

The transfer utility *globus-url-copy*, which is invoked in the data transfer DAG nodes generated by *gencdag*, requires its arguments a source and destination URL which contain an absolute pathname.

Job steps for cataloging entries in the replica catalog will be inserted into the DAG. These jobs will be executed at the SH (Submit Host). The planner will generate these job steps through its generalized replica catalog interface. There is no provision (currently) to remove files that have been moved to an SE by the planner, or to un-catalog such removed files. User must un-register such files manually by either invoking the replica-catalog script provided with VDS.

It is assumed that all files and directories needed for executable applications (with the exception of those defined by in and out files in the derivation) will be set up, manually, ahead of DAG execution, at each execution site. To determine the PFNs of executables, an exec-lfn hint can be used to supply the physical file name of the executable to be used on the CE. If no exec-lfn-hint is provided then the transformation name is mapped to a PFN via the tc.data file. It will be assumed that the executable has already been (manually) installed at this location as seen from the CE.

The life-cycle of files in storage elements is as follows. DVs contain output LFNs for new files to be created. Once the pool for a DAG to execute the DV in is chosen, gencdag translates all output LFNs within a DAG to a PFN.. After the job (DAG node) completes, the data transfer post-job for that DAG node transfers all output files to the output SE for the DAG, which is obtained from the “-o pool” argument to gencdag. (If no -o pool is specified, or if the -o pool is the same as the -p pool where the job was executed, then no post-job data transfer is done).

4.9 The Transformation Catalog

The transformation catalog is the text file “tc.data”. By default the transformation file is picked up from \$VDS_HOME/tc.data but the user can set it to any file in the \$VDS_HOME/etc/properties file.

Each line is a tuple (poolname, logical transformation, physical transformation, environment string).

pool is any valid location registered in the replica catalog providing computational and/or storage facilities. Special pool name “local” denotes the storage/compute element is the local machine and not a remote gatekeeper or pool.

logical transformation is the logical name of the transformation or executable

physical transformation is the actual path to the transformation which is mapped to the logical transformation.

environment string is used to specify all the environment variables Condor has to set in order to transformation to run on the execution pool. The environment variables can also be passed through the dax or the pool config file in xml format using the env namespace in profile element .If the same environment variable is specified in more than one place, then the following order of preference is followed property file , transformation catalog, pool config file and dax, with value specified in property file having the highest precedence and the ones in dax file the least precedence . Multiple environment variables can be defined by separating name=value pairs by using“;”. An empty value is specified using the word “null”.

Example tc.data file:

#poolname	logical transformation	physical transformation	environment string
isi	extract	/usr/local/bin/FrCopy	LIGO=/usr/ligo;PATH=/usr/bin
isi	concat	/usr/local/bin/FrCopy	LIGO=/usr/ligo;PATH=/usr/bin
isi	globus-url-copy	/usr/globus/bin/globus-url-copy	GLOBUS=/usr/globus2
local	GriphynRC	/usr/chimera/bin/griphynrc	VDS_HOME=griphynrc
isi	resample	/usr/local/bin/resample	LIGO=/usr/ligo;PATH=/usr/bin
isi	sft	/usr/local/ldas/bin/sft	LDAS=/usr/local/ldas
isi	deft	/usr/local/ldas/bin/sft	LDAS=/usr/local/ldas

Required entries in the Transformation Catalog

The following logical transformations need to be entered in the Transformation Catalog for Pegasus to generate a concrete DAG

GriphynRC	This logical transformation name refers to the default mechanism to access the Globus replica-catalog. The java client registers all the materialized data with their location as specified by the output pool. This script is to be run on the submit host site (the site where Condor-G runs). The pool for this entry needs to be the special pool handle "local" (without the quotes). The TC entry for this must specify the JAVA_HOME and the VDS_HOME environment variables for the submit host.
RLS_Client	This logical transformation is used in conjunction with the RLS replica catalog. A Java client rls-client registers all materialized data with their location as specified by the

	output pool. This script is to be run on the submit host site (the site where Condor-G runs). The pool for this entry needs to be the special pool handle "local" (without the quotes). The TC entry for this must specify the JAVA_HOME and the VDS_HOME environment variables for the submit host.
gsincftpget	Deprecated.
gsincftpput	Deprecated.
globus-url-copy	This is the logical transformation name that refers to the globus-url-client that is installed at the remote pool site. The globus-url-copy is the default transport. The TC entry for this must specify the GLOBUS_LOCATION and, unless linked statically, the LD_LIBRARY_PATH environment variables for the pool concerned.
transfer	This is the logical name of the executable which allows multiple transfer in one DAG job. Its current status is experimental. A sample implementation as C program which uses globus-url-copy internally is provided as \$VDS_HOME/bin/transfer.

Table 8: Implicit transformations in the TC.

4.9.1 The Transfer Mechanism

Currently, two kinds of staging mechanisms are supported. The so-called single file transfer generates one invocation of *globus-url-copy* for each staging request. This is the default, and also the current production recommendation. The other mechanism is capable of bundling multiple transfers with slightly improved reliability into one DAG job..

4.9.1.1 Single Transfers

As default transfer mechanism, the *globus-url-copy* client pulls data from the storage site or transfer the materialized data to a storage location. In order to determine the storage sites to stage-in data from, the replica catalog/Replica Location Service is queried by Pegasus. It returns the locations of data existence. One location is picked, and a stage-in job created. The storage site to which all the data is staged-out is specified as the output pool when invoking Pegasus. All storage sites must be gridftp enabled.

Pegasus looks for *globus-url-copy* clients under the logical name of *globus-url-copy* in the *tc.data* file. For the concrete planner to run successfully, the entry must exist in the transformation catalog, as described in section 4.9

The staging in and out of files works as follows

- A “stage in” node is added per data file per job to move in the required data for execution to the execution pool.
- Similarly, to stage materialized files to the output pool, a “stage out” node is added per output file per job to move the materialized data to the output pool if asked for.
- The registration for the materialized files is done per node, so that if a job has 4 output files, then 4 parallel nodes are added to transfer the files to the output pool, and one node which registers all the files in the replica catalog.

The above described procedure is the default mode of operation to achieve data transfers. Note the following limitation: All files are generated in the current working directory. Absolute logical filenames (starting with a slash) are not supported. Hierarchical logical filenames (containing slashes) are not supported.

The one-file-at-a-time transfer is the default mode of Pegasus. To enforce this kind of operation, you may set the property *vds.transfer.mode* to “multiple” (yes, to “multiple”).

4.9.1.2 Multiple Transfers

The second transfer mechanism makes use of the transfer executable (referred in the transformation catalog by the logical name of transfer). The transfer executable is a wrapper around *globus-url-copy*, which tries to run up to four parallel file transfers, each with 4 parallel streams, while occupying only one job on the scheduling system.

The semantics are similar to those of single transfers. The only difference being that the “stage in” and “stage out” nodes are per job instead of being per file per job as in the case of single transfers. In addition the placement of the transfer jobs are more sensible as compared to the single transfer mode. In this mode, explicit care is taken to ensure that duplicate transfer jobs are not inserted that end up transferring the same input file to the same location on the execution pool. This case would arise in case of single transfer if more than one job requiring the same input file are scheduled on the same execution pool by Pegasus. However, in the multiple transfer mode dependencies are added in such a manner that there is only one transfer node that is created and the jobs are dependant on that. A similar situation can arise in case of inter pool transfers, that is correctly handled in this mode and not in the Single Transfer mode.

A user may provide her own implementation of the (multiple file) transfer mechanism. The *gencdag* program will generate output matching to the specified API, if the multiple transfer option is activated. The API to the transfer program is as follows:

1. The first CLI argument is the base-URL prefix string. If this prefix is found in the URL list, it will be replaced by the mount-point as specified in the next CLI argument. This feature allows for optimization by changing 3rd-party-gsift-URL schemas into 2nd-party-transfer file schemas. If you want to avoid optimization, use a random string that is unlikely to be a URL prefix.
2. The second CLI argument is the mount-point to be used in a file-URL replacement for matched base-URL prefixes. This feature allows for optimization by changing 3rd-party-transfer gsift-URL schemas into 2nd-party-transfer file schemas.
3. The third CLI argument is optional. It is the name of a textual file that contains all source and destination URLs as specified below. If the third argument is not specified, *stdin* will be assumed as source of the URL list.

The URL list features an even number of textual lines. Each pair consists first of the source, and then of the destination URL. The line-separated approach was chosen, because some URLs may contain (illegal) white-spaces. All URLs must be in a format understood by *globus-url-copy* for the shipped implementation. The shipped *transfer* executable will dynamically invoke `$GLOBUS_LOCATION/bin/globus-url-copy`.

The GVDS provides an executable “transfer” that is part of the worker node package, which complies to the above API. Our transfer implementation calls *g-u-c* internally. Hence, the setup in the transformation catalog must reflect the setup for regular *globus-url-copy*, e.g. the environment variable setup. The provided transfer feature parallel transfers, parallel streams, and some limited retry capabilities.

The “transfer” executable will place less burden on the gatekeeper, and should behave more robust. But the “transfer” executable will also place a lot more burden on your gridftp servers, especially with parallel data streams, up to the point where they cannot serve requests decently. In a production environment that transfers many files, you may want to consider separating the gridftp server from the gatekeeper. Both services need to reside in the DMZ. However, the user can throttle the number of *g-u-c* processes spawned by the transfer executable, and the number of streams a *g-u-c* process uses to data transfer by setting the `vds.transfer.throttle.processes` and `vds.transfer.throttle.streams` respectively.

Note: With the advent of a multi-file capable *globus-url-copy*, the API *may* change slightly to match the requirements of *globus-url-copy*.

The multiple file transfers are an experimental new mode of Pegasus. To enforce this kind of operation, you may set the property `vds.transfer.mode` to “multiple” (yes, to “multiple”, because a single transfer job is generated to transfer multiple files).

4.10 Replica Registration

4.10.1 Using the Globus Replica Catalog

The Replica Catalog is a part of the Globus toolkit, which provides a logical filename to physical filename mapping. This is a one-to-many many mapping i.e. a given logical file can be stored at various physical locations (sites or pools). This mapping is used to resolve the logical names in the DAX file generated by the abstract planner. The data in the Replica Catalog can be organized in one or more logical collections. When Pegasus looks up a logical filename, the replica catalog is queried for its location to see whether it exists or needs to be materialized. More than one location can be returned for a particular logical file. If any of the locations returned is the same as the pool (the `re` attribute of the location object matches the execution pool) on which the dag is being executed, then that location is selected, otherwise, a random location is picked from the set of existing replicas.

It is possible that a file requested by the user has already been materialized, in which case no computation needs to take place. Thus the Replica Catalog is used to determine whether a particular job needs to be executed or not, leading to its removal from the DAG, as well as the recursive removal of previous jobs which were in the DAG to generate the data for the execution of that job. *In order to construct a concrete DAG, it is essential that the Replica Catalog be populated with data that represents the input data for the root nodes of the files of the abstract DAG – the inputs to the execution, otherwise, the data cannot be materialized.*

All the data that is materialized during the execution of the DAG is copied to the storage element on the output pool, if one was specified to Pegasus. The output data files are registered with the Replica Catalog for future use. The output data produced on the execution host also remains in place until removed by some external process.

For more details regarding Replica Catalog refer to [Getting started with the Globus Replica Catalog](http://www.globus.org/datagrid/deliverables/replicaGettingStarted.pdf) (<http://www.globus.org/datagrid/deliverables/replicaGettingStarted.pdf>). Examples of how to populate the replica catalog using the java client (that ships with this release) are given in the VDS man page for *replica-catalog* and the appendix of this user guide.

Note: Do not use the *globus-replica-catalog* client that comes with Globus to populate your replica catalog. For more details, refer to Section 12.

4.10.2 Using the Replica Location Service

The Globus Replica Location Service (RLS), <http://www.globus.org/rls>, the successor to the Globus Replica Catalog, maps logical file names to physical sites and physical file names in a scalable and fault-tolerant manner.

The RLS consists of at least one LRC (Local Replica Catalogs) reporting to one or more RLIs (Replica Location Index). As of now, we have tested the configuration in which many LRCs report to a single RLI. Ideally, each LRC should be responsible for data residing at one pool. However, same LRC can be used for more than one pool. This is possible by specifying different pool names while registering a mapping to the same LRC using the *rls-client*.

The RLS servers use a MySQL database in their backend. Thus, it is expected to better cope with large amounts of logical files.

To obtain a LFN to PFN mapping, one queries the RLI with the LFN. The RLI returns a list of LRCs that contain the requested mapping. In the next step, any of the LRCs from the previous result can be queried for the mapped PFN. Please note that the RLI does not contain the LFN to PFN mapping itself.

The RLS interface requires the Globus RLS client libraries to be installed at the submit host. The Java client dynamically probes for various flavors of the RLS client libraries, as the interaction goes through JNI. As of this writing, installation is not trivial.

To configure and seed your RLS server you need to do the following steps:

You need to define an attribute “pool” for the PFNs on each LRC. This is done with the following command.

```
globus-rls-cli attribute define "pool" "pfn" "string" rls://<rls-host-name>
```

The *globus-rls-cli* client can be used for bulk seeding of the RLS. Create a file “seedlrc” with the commands

```
create foo1 bar1
attribute add bar1 "pool" "pfn" "string" <pool-name>
create foo2 bar2-1
add foo2 bar2-2
add foo2 bar2-3
attribute add pfn2 "pool" "pfn" "string" <pool-name>
create lfn3 pfn3
attribute add pfn3 "pool" "pfn" "string" <pool-name>
```

and so on. Pipe or connect the *stdin* of *globus-rls-cli* to the file you just created:

```
cat "seedlrc" | globus-rls-cli rls://<rls-host-name>
```

4.10.3 Installing the Replica Location Service

The recipe below refers to an old version of RLS to be integrated into an old version of Globus. Nowadays, RLS has better integration into Globus, and can be installed using the Globus package manager. The only catch is that `JAVA_HOME` must be set and point to a valid JDK 1.4 installation.

There are multiple ways to install the RLS client tools. Pegasus interfaces through JNI with the C-native Globus libraries. The RLS client is not (yet) part of a standard Globus installation, nor a VDT installation.

If you are using VDT 1.1.6, you cannot extend it due to a malfunction in its SDK. For your convenience, ATLAS was so nice to provide the client libraries for a modern Linux system running VDT 1.1.6 in `$VDS_HOME/contrib/rls`.

If you have a Globus 2.0 installation (untested with 2.2), the following steps should be able to get you started. Please make sure that your Globus info packages were compiled with threading enabled!

1. Download the latest RLS client sources and untar in a convenient location.
2. Check and make sure that your `$GLOBUS_LOCATION` is set and valid.
3. Check and make sure that your `$JAVA_HOME` is set, valid, and points to a JDK 1.4.
4. Edit the `java/Makefile.in`. Search for the `^all-am` regular expression, and add:

```
testrls.class: testrls.java
<tab>@JAVA_HOME@/bin/javac -classpath rls.jar $^
```

```
RLSGui.class: RLSGui.java
<tab>@JAVA_HOME@/bin/javac -classpath rls.jar $^
```


5. Use most of the following options to configure as appropriate for your Globus installation:
 - `--with-jdk-path=$JAVA_HOME` should always be specified
 - `--with-flavor=gcc32` or `gcc32dgb`, either one or the other
 - `--enable-debug` if you used `gcc32dbg` above.
 - `--with-threads=pthreads` should always be specified.
6. Sometimes, it is necessary to adjust the `CPPFLAGS` in the generated Makefile, e.g.:


```
CPPFLAGS = -I$(GLOBUS_LOCATION)/include -I$(GLOBUS_LOCATION)/include/gcc32dbg
```

 Perhaps you may need to add also the include for `gcc32dbgpthr` if you used `gcc32dbg` above.
7. Try a “make” and “make install”. This will install the libraries, headers and C binaries.
8. If the Java installation produced failures, “chdir java” and try a “make install”.
9. Experienced admins may want to collect all `java/RLSGui*.class` files into a jar, and make this jar publicly available as a simple GUI into the RLS.

Recent versions of RLS are also installable with a recent version of the Globus Packaging Toolkit (GPT). In order to use the GPT, make sure that `$GPT_LOCATION` is set and valid.

4.10.4 Setting up your own RLS server

Please note that the RLS consists of two parts. The LRC is a local service that maps a given LFN onto what it calls a PFN, and possibly more attributes. The RLI is the index server, which only maintains a mapping from a given LFN to all LRC that know about this. Thus, the LRC must periodically talk to its RLI to provide  mappings. Both, LRC and RLI, may run with the same RLS server installation.

Before running the GVDS provided `rls-client`, make sure to set the properties `vds.replica.mode` and `vds.rls.url` correctly. In order to link the local LRC to the RLI, and prepare the RLS for GVDS replica data, follow these steps. We assume that both, your LRC and RLI are on the same local machine.

1. You need to ensure that your LRC is updating to the correct RLI. You can specify any RLI to send updates to. This can be done by using the following C command from the RLS server package:

```
globus-rls-admin -a rls://<rli-name> rls://<lrc-name>
```

where you need to replace `<rli-name>` with the hostname of your RLI and the `<lrc-name>` with the hostname of your LRC. If your RLS server is providing both services, this should be the same hostname.

2. The LRC sends periodic updates to the RLI. To force the LRC once to send an update, use:

```
globus-rls-admin -u rls://<lrc-name>
```

Again, the `<lrc-name>` is the hostname of your local LRC.

3. To add the necessary additional column for Pegasus to turn the generic RLS into a GVDS-specific RLS, you need to invoke the RLS client manually once for each LRC:

```
globus-rls-cli attribute define "pool" "pfn" "string" \
  rls://<lrc-name>
```

The `<lrc-name>` is to be substituted for the LRC's host name. Please note, this step needs to be done only once each LRC to prepare it for GVDS data. Refer to section 11.2 for details on the usage.

4. To successfully use the RLS, each pool, including the compute pools, is required to have an entry in the RLS. Thus, it is recommended to use the GVDS application *rls-client* to enter a dummy for each pool:

```
rls-client --lrc rls://<lrc-name> --verbose --delimiter @ \
--mappings=dummy,gsiftp://test.uchicago.edu/home/me/dummy \
--pool uchicago
```

The above string will enter a logical file “dummy” with the (fake) physical location in the UofC under a pool handle “uchicago”. Please note that the chosen gsiftp-URL and path to the (non-existent) dummy file needs to match your entries in *pool.config* for the chosen pool. Repeat for each pool you intend to use.

4.11 Pool Configuration using MDS

4.11.1 XML Based pool configuration file

As described earlier in section 4.5 the Pegasus planner takes in the pool configuration file, that gives it the setup of the remote execution pools. This pool configuration file can either be textual (as in section 4.5 that would be deprecated soon) or can be in xml format that can be generated by querying the Globus MDS. This section explains how to generate the new xml based format and its semantics. The xml based pool configuration file is similar to the textual format but in addition it has support for multiple jobmanagers, multiple grid ftp servers and profile information per pool, that was missing in the textual format.

The information about each pool is generated by running a *gvds-infoprovider* script through MDS, on each pool. The *gvds-infoprovider* script is available in *VDS_HOME/libexec* directory. Each instance of the script on the remote pool takes in a text file as input known as *gvds.pool.config* file. Each *gvds.pool.config* file is to contain information about one execution pool that is piped through the GRIS on that pool to the central GIIS that you have setup. A sample *gvds.pool.config* file is distributed with the VDS source and can be found at *\$VDS_HOME/etc/sample.gvds.pool.config*

A sample *gvds* pool config file is written below

```
#####
#          GVDS POOL CONFIGURATION          #
#####
gvds.pool.id : isi
gvds.pool.lrc : rls://smarty.isi.edu
gvds.pool.lrc : rls://dc-user.isi.edu
gvds.pool.lrc : rls://dc-user.isi.edu
gvds.pool.lrc : rls://druid.isi.edu
gvds.pool.gridftp : gsiftp://smarty.isi.edu/smarty/sources/@2.2.4
gvds.pool.gridftp : gsiftp://birdie.isi.edu/nfs/asd2/gmehta/GRIPHYN@2.2.2
gvds.pool.universe : vanilla@birdie.isi.edu/jobmanager-condor@2.2.4
gvds.pool.universe : vanilla@smarty.isi.edu/jobmanager-condor2@2.2.2
gvds.pool.universe : standard@birdie.isi.edu/jobmanager-condor@2.2.4
gvds.pool.universe : transfer@birdie.isi.edu/jobmanager-fork@2.2.4
gvds.pool.universe : ldas@smarty.isi.edu/jobmanager-ldas@2.2.4
gvds.pool.universe : globus@smarty.isi.edu/jobmanager@2.2.4
gvds.pool.universe : pbs@smarty.isi.edu/jobmanager-pbs@2.2.4
gvds.pool.gridlaunch : /users/bin/grid-launch
gvds.pool.workdir : /tmp/
gvds.pool.profile : env@GLOBUS_LOCATION@/smarty/gt2.2.4
gvds.pool.profile : env@JAVA\_HOME@/smarty/jdk1.4.1
gvds.pool.profile : vds@VDS_HOME@/nfs/asd2/gmehta/vds
```

As evident from above, the user can specify multiple LRCs, multiple gridftp servers and multiple jobmanagers per universe for any execution pool. Each remote execution pool is supposed to run one GRIS that is responsible for that pool, and for piping its pool configuration to the central GIIS which Pegasus queries to generate the XML based pool config file.

4.11.2 Setup of an MDS-driven configuration

The MDS that need to be installed should be version 2.4 or higher. It should have specifically the fix for the missing information that was prevalent in version 2.2. The MDS that ships with VDT is not fully functional, and hence the user needs to install MDS in his VDT install directory fresh after installing the VDT. A pacman cache is provided at ISI, that installs a version of MDS 2.4 that is known to work with the infoproviders.

Before proceeding with the installation of MDS, please take a backup of \$VDT_LOCATION/setup.csh as that is changed while installing MDS. The backup of csh can be copied back, once the installation is done.

1. source \$VDT_LOCATION/setup.csh . \$VDT_LOCATION is the base directory where VDT is installed. This script sets environment variables like VDS_HOME, where VDS (Pegasus/Chimera) was installed.

2. Install the pacman cache:

```
pacman -get ISI:MDS24-ligo
```

This installs a patched version of MDS in \$VDT_LOCATION/MDS24.

3. As superuser (root), install the start-up script

```
cp -r $VDT_LOCATION/MDS24/sbin/SXXgris /etc/rc.d/init.d/gris
```

4. Each site needs to have a configuration pool-file for Pegasus, where the administrator sets up the pool information that Pegasus uses. The format of gvds.pool.config - as described in the previous section - should be kept in the \$VDS_HOME/etc directory.
5. Edit the file \$VDT_LOCATION/MDS24/etc/ grid-info-resource-ldif.conf. The user needs to add the following lines in the end

```
#Generate the gvds pool information for Pegasus Montage every 30 second
dn: Gvds-Software-deployment=Gvds,Mds-Host-hn=$HOSTNAME,Mds-vo-
name=local,o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: $VDS_HOME/libexec/
base: gvds-infoprovider
args: -dn Gvds-Vo-name=$VO_NAME,Mds-Host-hn=$HOSTNAME,Mds-Vo-
name=local,o=Grid -f $VDS_HOME/etc/gvds.pool.config
cachetime: 30
timelimit: 20
sizelimit: 100
```

All the environment variable above need to be replaced by their values while writing in grid-info-resource-ldif.conf.

\$HOSTNAME = the fqdn of the host e.g. sukhna.isi.edu

\$VDS_HOME = the directory where VDS is installed on remote execution pool.

\$VO_NAME = the name of your VO e.g LIGO.

6. In \$VDT_LOCATION/MDS24/etc/grid-info-slapd.conf add the following lines

```
include $VDS_HOME/etc/gvds-pool-info.schema
include $VDS_HOME/etc/grid-info-gridftp-bandwidth.schema
```

The environment variable needs to be replaced with it's correct value.

7. For the machine that acts as your central GIIS, the following is needed. In `$VDT_LOCATION/MDS24/etc/grid-info-slapd.conf`, edit the entry for the database called "giis". Below the line

```
database      giis
```

replace the line

```
suffix        "XXX"
```

with

```
suffix        "Mds-Vo-name=SC, o=Grid"
```

By default, the suffix is

```
Mds-Vo-name=site, o=grid
```

8. For the machines that act as a GRIS, the user needs to specify which GIIS to report to. This is specified in the `VDT_LOCATION/MDS24/etc/grid-info-resource-register.conf` file. The `dn` entry for the second `ldif` record needs to be changed to

```
dn: Mds-Vo-Op-name=register, Mds-Vo-name=SC, o=grid
```

and `reghn` is the host that is your central GIIS.

A sample `grid-info-resource-register.conf` is enclosed below. The GRIS is running on `pisa.isi.edu` and reporting to the GIIS running at `sukhna.isi.edu`:

```
dn: Mds-Vo-Op-name=register, Mds-Vo-name=SC, o=grid
regtype: mdsreg2
reghn: sukhna.isi.edu
regport: 2135
regperiod: 300
type: ldap
hn: pisa.isi.edu
port: 2135
rootdn: Mds-Vo-name=local, o=grid
ttl: 1200
timeout: 20
mode: cachedump
cachettl: 30
bindmethod: ANONYM-ONLY
```

9. Start the GRIS . It does not need to be started as root:

```
/etc/rc.d/init.d/gris start
```

4.11.3 Generation of the pool config file from MDS

The pool configuration file is generated by querying the central GIIS for your grid setup using `genpoolconfig` utility that is distributed with VDS. It is available by default in `$VDS_HOME/bin` directory.

Before running the tool, the user needs to ensure that the GRIS is running on the various machines and reporting to the GIIS. One can use a LDAP browser like gq to check if GRISes are reporting to the GIIS correctly or not.

The *genpoolconfig* tool queries the central GIIS, and generates the pool configuration file. It needs the following two properties to be specified, either at the command line or in the user properties file.

```
vds.giis.host      sukhna.isi.edu:2135
vds.giis.dn        Mds-Vo-name=SC, o=grid
```

The first line specifies the host at which the GIIS is running, including the port number. By default, the GIIS runs at port 2135. This is same as the suffix specified in step 7 of the previous section of setting up the MDS. To generate the pool config file, execute *genpoolconfig* thus:

```
genpoolconfig --output pool.config.xml
```

Or, with properties specified at the command line (yellowed to protect whitespaces):

```
genpoolconfig -Dvds.giis.host=sukhna.isi.edu:2135 \
-Dvds.giis.dn=Mds-Vo-name=SC,o=grid --output pool.config.xml
```

5 Command Line Toolkit

The primary interface to the VDS is a set of simple command line interface tools. Each VDS command is either a shell script or a batch script that starts Java with the correct class, and passes the environment and command line arguments. Most of the commands operate on one or more TR or DV definitions.

A XML database file, which we will refer to throughout this section as the VDB, will be created, if it does not already exist.

The command-line tools are designed as simple filters, moving VDLx or VDL2 definitions to and from files, and listing definitions in VDLx, VDLt, or simple text list formats. The tools are designed so that they can be composed into more powerful tools or applications, both as shell scripts and via invocation from other application frameworks (dynamic web pages, or applications in virtually any language that can invoke a shell).

All VDS commands return an exit code of 0 for successful execution. If any failures occur, the commands catch any Java RuntimeException, show the necessary stack trace and message, clean up, and exit with an exit code of 1.

The command line toolkit consists of the following commands:

vdlt2vdlx	Compile VDLt into VDLx format
vdlx2vdlt	Decompile VDLx into VDLt text
updatevdc	Insert or change VDL definitions in the VDC
deletevdc	Delete one or more TR or DV definition from the VDC
searchvdc	Find and list VDL definitions from the VDC
gendax	Generate an abstract DAG (DAX) as XML by requesting an LFN or DV.
gencdag	Convert an abstract DAG into a concrete Condor DAG ready to submit
shplanner	A simple shell planner which converts an abstract DAG for local execution.
replica-catalog	Access the replica catalog service for the Globus 2 replica catalog.

<code>rls-client</code>	Access the replica location service for the RLS.
<code>rls-query-client</code>	Allows to query the RLS and include pool information into the output.
<code>vds-version</code>	Reports the version number of the Virtual Data System.
<code>exitcode</code>	Obtains the remote job exit code and manages provenance tracking.

Table 9: VDS CLI applications.

<code>archstart</code>	Script to select the appropriate binary platform
<code>keg</code>	kanonical executable for grids – remote site test program
<code>kickstart</code>	Grid launcher, provides provenance tracking information
<code>*-free</code>	Programlet to determine the memory size of a machine
<code>transfer</code>	Multi-file wrapper for globus-url-copy

Table 10: VDS helper applications.

Please refer to section 4.2 for the necessary properties that the command-line tools will employ. The properties allow the user to override several default locations, e.g. for the *tc.data* and *pool.config* file, as well as default locations of Chimera-supplied XML schemata. If the corresponding properties *vds.schema.vdl* and *vds.schema.dax* are not set, the schema files in their default location in `$VDS_HOME/etc` are used.

Almost all VDS commands allow an experienced user to override properties from the properties file. For this purpose, the `-D` command line option may be specified one or more times, once of each property. Caveat: The `-D` option(s) must be specified first in the command line string. With the help of this property extension mechanism, the default location of the property file may be overwritten, the location of the VDS system modified or just some simple options modified.

5.1 Conversion between VDLt and VDLx

Two applications, *vdlt2xml* and *xml2vdl*, are provided to convert between VDLx and VDLt. Each application takes either zero, one or two command line interface arguments. Called with zero arguments, it will read from stdin and write to stdout. Called with one argument, it will read from the specified file, and write to stdout. Called with two arguments, it will read from the first file, and write to the second file. If the second file already exists, it will be overwritten.

```

vdlt2vdlx [-Dprop [...]] < VDLt > VDLx
vdlt2vdlx [-Dprop [...]] VDLt > VDLx
vdlt2vdlx [-Dprop [...]] VDLt VDLx

vdlx2vdlt [-Dprop [...]] < VDLx > VDLt
vdlx2vdlt [-Dprop [...]] VDLx > VDLt
vdlx2vdlt [-Dprop [...]] VDLx VDLt

```

A note concerning the commutative-ness: The XML version of the document is slightly more expressive than the textual version. For this reason, a conversion of a document A from VDLx into document B in VDLt might lose some non-essential information. Thus, after conversion of B back into VDLx as document C, there is a difference between A and C. A further conversion of C into VDLt document D will display no differences between B and D. Further conversions back and forth will not display any more differences – the process becomes stable and is commutative after any initial loss from VDLx to VDLt.

5.2 Creating, adding, and updating definitions

```
insertvdc [-Dprop [...]] VDLx ... # only insert, no updates
updatevdc [-Dprop [...]] VDLx ... # insert or update
```

The VDC database, which we will refer to throughout this section as the VDDb, will be accessed. By default, the *VDDb* from the user's properties files will be taken. This VDDb specification can be overwritten using appropriate properties.

An arbitrary number, but at least one, VDLx document can be added to the database. The *only* way to get definitions into the database is via VDLx. If a user wants to use VDLt, the converter from the previous section has to process files first.

Using `insertvdc`, new definitions will be added to the database. Definitions that already exists in the database will be ignored.

Using `updatevdc`, any definition (that is, either a TR or a DV) in the VDLx file(s) will overwrite a known definition with matching attributes that constitute the primary key. If the definition did not previously exist, it will be added.

The insertion and update commands are the preferred way of getting information into the persistent storage. As the current implementation allows to employ a VDLx file as database, you can “create” a new file-based database by copying your VDLx file with the following simple recipe:

```
cp vdlxfile some/file.db.0
echo "0" > some/file.db.nr
```

After executing this recipe, you can point your properties to “some/file.db”, and thus work on that particular file-based database. Please note that the above shortcut will only work with the initial releases of Chimera. Future versions of Chimera will employ other methods of persistent definitions storage, and thus the insert and update tools must be used to enter data.

5.3 Deleting definitions

```
deletevdc [-Dprop [...]] [-t tr|dv] [-n ns] [-i name] [-v version]
```

The deletion offers a form of searching and multi-deletions, though joker characters are not allowed. Future versions of Chimera which use a different persistent storage may introduce permissible joker characters.

Not finding an item to delete is not an error, still returns a zero exit code.

Deletion works on an existing database. It does not work on VDLx. If VDLx files need to be worked upon, they need to be converted into a database first, and results may be converted back into VDLx. With the current file-based database, the “conversion” can be short-cut as described in the previous section.

From one to four of the mandatory options `t`, `n`, `i`, or `v` must be specified:

- The “-t” option specifies whether to limit deletions to just TR, or just DV (case insensitive). If this option is not specified, deletions will work on both kinds of definitions.
- The “-n ns” option defines the namespace to search for a TR or DV. If this option is not specified, any namespace will be matched. There is no default namespace.
- The “-i name” option defines the identifier (name) of the definition. A missing identifier option means that the definition name is not part of the search. Hence, any named object will be deleted.
- The “-v version” option works similar as “-n” and “-i”. It applies to the version of the definition. A version may be optional for both, missing version flag will be taken as wildcard (Yong, as above). (Note that versions are simply string that must match exactly).

5.4 Searching, listing and dumping

Searches in the database can show their results in one of three modes:

1. Output can be formatted textually similar to SQL query output (option `-l n`),
2. Results can be described employing VDLx (option `-l x`), or
3. Output can be described employing VDLt (option `-l t`).

By default, the textual tabular output is chosen, thus the “`-l n`” option is implicit.

```
searchvdc [-Dprop [...]] [-l x|t|n ] [-o out] [-t TR|DV] [-n ns] [-i name] [-v
version]
searchvdc [-Dprop [...]] [-l x|t|n ] [-o out] [-t i|o|io] [-f lfn]
```

The “`-d dbprefix`” option works in the usual fashion, allowing to specify an alternative database. Results are put onto stdout by default. The `-o` option allows specifying an alternative file to put the results into. The “`-l`” option specifies the format to use for showing results.

The search tool has two ways to be invoked: It can either search for a definition, or it can search for a logical filename. Thus, the parsing of the “`-t`” option depends on the context.

In order to do simple searches for definitions, `app4` works with the same set of command line interface arguments as deletions in the previous section with one difference: It is permissible to specify none of the namespace, name, and version options. If none of `-t -n -I -v` are specified, then all definitions in the db are dumped.

In the first form, from one to four of the mandatory options `t`, `n`, `i`, or `v` must be specified:

- The “`-t`” option specifies whether to limit deletions to just TR, or just DV (case insensitive). If this option is not specified, deletions will work on both kinds of definitions.
- The “`-n ns`” option defines the namespace to search for a TR or DV. If this option is not specified, any namespace will be matched. There is no default namespace.
- The “`-i name`” option defines the identifier (name) of the definition. A missing identifier option means that the definition name is not part of the search. Hence, any named object will be deleted.
- The “`-v version`” option works similar as “`-n`” and “`-i`”. It applies to the version of the definition. A version may be optional for both, missing version flag will be taken as wildcard (Yong, as above). (Note that versions are simply string that must match exactly).

In the second form, one or both of the mandatory options `-f` and `-t` must be specified:

- The “`-t`” option specifies the file-type as either *input*, *output* or *inout* file, using `i`, `o` and `io` respectively.
- The “`-f`” option specifies the filename.

Not finding anything to match the search is no error. Thus, the exit code will still be zero.

5.5 Producing an abstract DAG in XML (“DAX”)

```
gendax [-Dprop [...]] [-o outfile] [-l label] -n ns -i name -v version
gendax [-Dprop [...]] [-o outfile] [-l label] -f f1[,f2[,...]] -D d1[,d2[,...]]
```

A DAX (which stands for “abstract DAG in XML”) can be requested in two ways. The user may either request that all matching derivations of a set be produced (case 1, above). In this case, the description allows wildcard search by not providing values. If multiple matches are found, all derivations will be produced.

A user may also request that one or more logical filenames be produced as output of a derivation (case 2, above). Case 2 allows mixing in derivations descriptions as namespace::name:version. The derivation must be fully qualified, and does not allow wildcard matching.

The options `-D` allows for a comma-separated list of fully-qualified derivations to produce. The `-f` option allows for a comma-separated list of logical filenames to produce. The `-f` and `-D` options may be mixed, and used multiple times. In essence, it is possible to request a “forest”, or disconnected graph, in a single request, and with a single DAX output.

A recent addition allows to overcome the commandline length limitations. The flag `--dvlist` takes the name of a file, which contains derivation names similar to `-D`, one per line. Empty lines and comment lines (`#`) are ignored. The `--filelist` flag takes the name of a file, which contains LFNs, one per line. Empty lines and comment lines (`#`) are ignored. The flags `-f`, `-D`, `--dvlist` and `--filelist` may be arbitrarily mixed.

If there are multiple matches for a derivation, or there are multiple derivations that can produce the requested logical file, the first encountered is generated.

The output is either a DAX document on stdout, or, in the presence of the `-o fn` command line interface argument, the DAX dumped into the specified file. If the file is already present, it will be overwritten.

Future approach: The DAX file is a basename, and alternatives from ambiguity resolution will be written into files `DAX.0.dax ... DAX.n.dax`. In the current version though, only one DAX will be produced in file `DAX.dax`. Decisions on alternatives will be done on a first-found basis.

The `-l` option allows to name a produced DAX. The label will be used by the concrete planner to generate the name of the DAGMan `.dag` file, and to mark the workflow with additional Condor ClassAd attributes. Pooling a workflow allows to select jobs that belong to one workflow, if multiple workflows are concurrently active. If `-l` is not specified the label “test” is produced.

If the extent of the search through the VDC needs to be limited, the `-maxdepth` option can be used. By default, the search depth is virtually unlimited. Future versions of `gendax` are going to allow limits based on metadata information.

5.6 Running the shell planner

The shell planner allows to test the Chimera system without requiring a grid to run this. It serves as first litmus test to check, if a user application will run with Chimera at all. Please note that the shell planner uses the local file-based stand-in for a replica catalog. It also makes use of the transformation catalog, but only uses the “local” pool entries.

```
shplanner [-Dprop [...]] [-b] [-n] [-d dir] daxfile
```

The shell planner reads a DAX file, which is the only mandatory input. Results are produced into a directory `dir` which defaults to “test”. The `-d` option allows to overwrite it. The `-n` mode works without updating the local file replica catalog.

By default, the shell planner runs in make mode. It will check the existence of files with the replica catalog and in the file system. The `-b` mode is a build mode, which will always produce the full production, regardless of file existence – existing files will be overwritten.

In the directory `dir` several files will be produced. For each job in the DAG there is a shell script which corresponds to the transformation name and DAX-id of the job. A master script which takes its name from the DAX-label controls the job scripts. For each job, list files help with the update of the stand-in replica catalog.

To run you shell plan, change into the directory `dir` and invoke the master shell script.

5.7 Supplementing the remote start-up

The tool `kickstart` is the only tool in the virtual data suite that runs on the remote pool.

Kickstart tries to run an executable in a more uniform environment with less setup hassles. Kickstart is a light-weight program that connects the *stdin*, *stdout* and *stderr* file handles, if the transformation to run needs to capture any of those.

Sitting between the remote scheduler and the executable, it is possible for kickstart to gather additional information about the executable run-time behavior, including the exit status of Globus jobs. This information is important for Chimera invocation tracking as well as to Condor DAGMan's awareness of Globus job failures.

Kickstart allows the optional execution of jobs before and after the main application job that run independent of the main application job. See the manual page for details about this feature.

All jobs with relative path specifications to the application are part of search relative to the current working directory (yes, this is unsafe), and by prepending each component from the `PATH` environment variable. The first match is used. Jobs that use absolute pathnames, starting in a slash, are exempt.

Kickstart rewrites the command line of any job (pre, post and main) with variable substitutions from UNIX environment variables. See the manual page for details on this feature.

```
kickstart [-n tr] [-N dv] [-i asi] [-o aso] [-e ase] [-l log] app [appflags]
```

Usually, you will only need to install kickstart in the remote pool, and tell its location in the transformation catalog. Pegasus will pick up things from there.

5.8 Obtaining remote job failure status and the provenance trail

Exitcode –TBD.

5.8.1 Failing workflows on remote job failure

Uses exitcode and property `vds.exitcode.mode`, requires kickstart and bug #931 fixes – TBD.


5.9 Requesting a Concrete DAG

5.9.1 Running the Pegasus concrete planner

```
gencdag --dax <dax file> --p <comma separated list of execution pools>
      [--dir <dir for o/p files>] [--submit] [--o <outputpool>]
      [--force] [--randomdir] [--help] [--verbose]
```

The Pegasus `gencdag` command takes in as input the DAX generated by the Chimera Abstract Planner. It generates the concrete dag in the form of condor submit files, which can then be submitted to one or more execution pools for execution.

Pegasus ensures that all the data that is required for the execution of the dag on the execution pool is transferred to. It achieves that by adding transfer nodes at appropriate points in the dag, resulting from the querying a replica catalog.

Pegasus  tries to reduce the dag by deleting the jobs whose output files have been found in some location in the Replica Catalog or in the Replica Location Mechanism unless force option is specified in which case the reduction step is skipped. Currently, no cost metrics are used, however preference is given to a location corresponding to the execution pool.

Pegasus can also add nodes to transfer all the materialized files to an output pool. The location on the output pool is determined by consulting the `pool.config` file. The path to these files is obtained from the `vds.pool.file` property value.

In addition the user can force Pegasus to execute the jobs in a random directory on the remote execution pools by giving the `randomdir` option at runtime. With this, create directory nodes are placed in the workflow that creates the random directory in the execution directory on the pool, that is determined from the pool configuration files. These create directory jobs refer to a logical transformation “`makedir`”, that ships with the VDS distribution. Eventually, this executable only would be called to cleanup these random directories.

For more details refer to the manpages.

5.9.2 Naming Conventions for the Condor files

The names of the condor submit files corresponding to the jobs which are to be executed are made up by concatenating the logical name of the job and it’s job id in the Abstract Dag (xml file). If the job name is `extract` and its id is `ID0001`, then the name of the submit file is `extract_copy_ID0001.sub`.

The transfer nodes which are added to transfer the files found in the Replica Catalog or Replica Location Service to the execution pool, are named as **`rc_tx_+jobname+_+counter+.sub`**, where counter varies from 0 to n-1, where n is the number of files for that job. *e.g* if there is a job whose corresponding submit file is `extract_ID0001.sub`, and it has 3 input files, then the transfer nodes for transferring the input files to the execution pool for that job would be named as `ip_tx_extract_ID0001_0.sub`, through `ip_tx_extract_ID0001_2.sub`.

The transfer nodes which are added to transfer the files in between the execution pools, are named as **`inter_tx_+jobname+_+counter+.sub`** where counter varies from 0 to n-1, where n is the number of output files of parent jobs that have to be transferred to it *e.g* if there is a job whose corresponding submit file is `concat_ID0002.sub`, and has two parents which are executed on different execution pools and generate two output files each, then the interpool transfer nodes would be named as `inter_tx_concatID0002_0.sub` through `inter_tx_concatID0002_3.sub`.

The replica nodes which are added if an output pool is specified, the naming schema convention is:

`new_rc_tx_+jobname+_+counter+.sub` for the replica node which transfers the output file to the output pool, and

`new_rc_register_+jobname+_+0.sub` for the replica node which registers the output file in the Replica Catalog.

For example, if we have a job whose submit file is `extract_ID0001.sub`, and it has five output files, then the submit files for the replica nodes transferring the output files for this job to the output pool would be `new_rc_tx_extract_ID0001_0.sub` through `new_rc_register_extract_ID0001_0.sub`.

If one gives the `randomdir` option to the Pegasus planner, then the nodes creating the random directories on the remote pool are placed. For these create directory nodes, the naming schema followed is **`pool name+_+_create_dir.sub`**. For example, the job that creates the random directory on the execution pool named `isi` would be named as `isi_create_dir.sub`.

5.9.3 Running the concrete DAG

The `gencdag` command submits the dag by default to the local CondorG which schedules the jobs to the globus-gatekeepers running on various pools. The gatekeepers submit the jobs via `jobmanager-condor` to the underlying condor pool. One has to look at the Condor logs to determine whether the jobs have executed correctly or not.

In later releases we will incorporate the parsing of the Condor log files to determine the successful completion of a job. By default, Pegasus simply generates the submit files without running them. *Please remember to generate your proxy using **grid-proxy-init** before running the Pegasus concrete Planner.*

The jobs are submitted to the local CondorG which schedules it to a condor-jobmanager running on the pool. This condor-jobmanager then schedules the jobs onto the underlying Condor pool. To determine whether the jobs have executed correctly or not, the user needs to view the Condor logs.

The user can also run things on the local CondorG (i.e. the submit host) by specifying the execution pool to be local, and making the corresponding entries in the pool.config file and the transformation catalog.

6 Examples

Some simple examples are part of the VDS base integrations, please refer to section 8.2.

6.1 The Shell Planner

The shell planner is designed to help you run Chimera derivations on a single local host, without using any grid software. It is useful for initial debugging of applications and for diagnosing if specific errors are caused by the grid or by Chimera. This section contains a brief recipe for running the shell planner.

1. Log onto your Linux system.
2. Ensure that your JAVA_HOME environment variable is set correctly.
3. Ensure that your Java is version 1.4, e.g. `${JAVA_HOME}/bin/java -version`
4. Download the latest tarball from <http://www.griphyn.org/workspace/VDS/snapshots.php>
5. Unpack and change into VDS directory .
6. Source the setup-user-env.* script suitable for your shell flavor (either Bourne or C). Unfortunately, this process is not foolproof: Some runtime environments require that you set the VDS_HOME variable to your current directory first.
7. Run the “test-version” program to check your runtime environment. If this program fails for any reason, you will be unable to run Chimera successfully. Do not continue with this list until *test-version* runs successfully.
8. Change into some work directory.
9. Copy the hello world example into a file hw.vdl:

```
TR hw( output file )
{
    argument = "Hello world!";
    argument stdout = ${file};
}
DV dl->hw( file=@{output:"out.txt"} );
```

10. Convert into XML: `vdlt2vdlx hw.vdl hw.xml`
11. If this is a pristine new installation, make sure to remove leftovers:


```
rm -f $VDS_HOME/etc/pool.config $VDS_HOME/var/tc.data
```
12. Insert into a local database: `insertvdc -d my.db hw.xml`
13. Generate the DAX file: `gendax -d my.db -l hw -o hw.dax -f out.txt`

14. Create a first TC: `echo -e "local\thw\t/bin/echo\tnull" > $VDS_HOME/var/tc.data`
15. Empty your RC: `cp /dev/null $VDS_HOME/var/rc.data`
16. Run the shell planner: `shplanner -o hw hw.dax`
17. Run the shell plan: `cd hw; ./hw.sh`
18. View the output: `cat out.txt`

These first steps should be executed to make extra sure that the Chimera part of your installation works. Ensuring that the grid infrastructure works for you is a lot more challenging due to the complex middleware. You might want to refer to section 8.1 to start verifying the grid infrastructure.

7 Documentation and some Frequently Asked Questions

All command line tools have appropriate manual pages available. Please refer to `$VDS_HOME/man/man1` for the nroff pages. A PostScript™, HTML and textual version can be found in `$VDS_HOME/man`.

All Java classes that constitute VDS are documented using the *javadoc* utility. You can point your browser to `$VDS_HOME/doc/javadoc/index.html` to start browsing the class documentation.

The XML schemata are documented by XMLSpy™. You can find the HTML documentation as well as the Word™ documentation in `$VDS_HOME/doc/schemas/<version>`.

In a demanding production environment, it is to be expected that various bugs and misbehaviors in the underlying middleware systems are discovered. The following Globus errors cause jobs to be held by Condor-G. Versions of Condor more recent than 6.4.4 allow a periodic retry which help overcome transient race conditions.

As of this writing, the following Globus errors were discovered as cause for Condor to hold jobs:

Globus error 17	The job failed when the job manager attempted to run it
Globus error 24	The job manager detected an invalid script response
Globus error 69	The job manager failed to create the temporary stdout filename
Globus error 73	The job manager failed to open stdout
Globus error 120	The job manager restart attempt failed
Globus error 121	The job state file doesn't exist
Globus error 129	The standard output/error size is different

The errors 69, 121 and 129 are usually transient, and will benefit from the retry feature – which is not yet part of VDS. The other errors usually require user intervention.

A couple of Globus errors allow for simple fixes:

Globus error 10	Data transfer to the server failed
If you use the <code>GLOBUS_TCP_PORT_RANGE</code> , most likely you ran out of sockets or ports for the firewalled host. To fix, extend the port range on the firewall.	
Globus error 132	The job was not submitted by the original jobmanager
Most likely, you access a pool with multiple jobmanagers, and you configured the wrong one. Configure the correct jobmanager, e.g. use <i>jobmanager-condor-INTEL-LINUX</i> instead of <i>jobmanager-condor</i> . Unfortunately, you will need to re-run the concrete planning stage.	

PORT command refused

Your staging failed despite ftpaccess being correct. Globus 2.0 is known to ignore the `-G` option to its gridftp server. Thus, it cannot determine the location where Globus is installed, and where it should look for configuration scripts. The simplest fix is to symbolically link the file `/etc/ftpaccess` to your Globus installation's `ftpaccess` file. Better fixes to run the `env` application from your `inetd`, and set the `GLOBUS_LOCATION` variable before running the gridftp server. The best solution is to apply the appropriate Globus patches.

Connection refused

Your staging failed, because many simultaneous gridftp transfers are incorrectly detected by your `inetd` as denial-of-service attack. Some daemons have a rate delimiter built-in. Please refer to the documentation for your `inetd.conf` on how to increase upon the default rate.

If you prefer to access VDS through its Java programming API, the most up-to-date documentation can be found in `$VDS_HOME/doc/javadoc/index.html`. Occasionally, the online version at <http://www.griphyn.org/workspace/VDS/javadoc/> is updated from our most recent version.

7.1 Before you shout help

There are a couple of things the user can do to help the developers narrow down problems. Please remember that you are usually working in a grid environment: hosts, routes and other network elements frequently go up and down. Many problems that you may be experiencing are transient in nature.

Furthermore, you are dealing with a multi-layered complex system. For debugging purposes it is easier, to work from the bottom up, and by-and-by add complexity. Often, even when you are suspecting that the GVDS is broken, in fact, some of the supporting fabric is experiencing problems.

If any of the simple tests stated in the subsections below fails, you will not be able to run the GriPhyN Virtual Data System, because something in the fabric is broken. Attempts to run GVDS jobs will be futile effort, until the broken fabric is fixed. The Condor and Globus documentation should help you identify and fix the problem.

7.1.1 Verify that you are permitted on the remote site

If you experience problems with a remote pool, the first thing to check is, if your certificate is being accepted on the remote site, including an example for a successful run. Please replace the string “<host>” with the fully-qualified hostname of the remote gatekeeper host:

```
globusrun -a -r <host>/jobmanager-fork
```

```
GRAM Authentication test successful
```

What you type is in bold, the expected response in regular font. The reasons your certificate is rejected can vary:

- You forgot run initialize your proxy certificate.
- There could be a mistake in the grid mapfile, when the administrator added your user certificate.
- Your certificate may not be known.
- The remote site does not have a copy of the CA-certificate of the CA which signed your user certificate.
- The CA certificate installed on the gatekeeper has expired.
- You are missing a CA certificate to contact the remote host (e.g. CERN).

- There may not be a jobmanager-fork available.
- The clocks between the two hosts are off by more than five minutes (when converted into UTC).

GSI errors are unfortunately unilluminative, and require either some degree of intuition to get to the bottom of, or a degree of perseverance.

7.1.2 Verify that the services are up, running and accessible

Occasionally a system administrator forgets to set up the correct path to the shared libraries required by Globus. While there are several ways to set up this facility, details are up to the site's policy. You can check that the services are accessible to you, and neither blocked nor failing by trying to telnet to the gatekeeper and gridftp server. The following instruction checks the availability of a gatekeeper:

```
telnet <gatekeeper> 2119
Trying <dotted-quad>...
Connected to <host>.
Escape character is '^]'.
^]
telnet> close
Connection closed.
```

Again, what you type is in bold, and the expected responses are in regular print. Note that the escape character “^]” for telnet is usually Ctrl+5. If you do not see the above responses, but instead something about a missing shared library, the site administrator needs to set up the LD_LIBRARY_PATH.

To check the availability of a gridftp service, try the following instructions:

```
telnet <gridftpserver> 2811
Trying <dotted-quad>...
Connected to <host>.
Escape character is '^]'.
220 <host> FTP server (GridFTP Server 1.5 [GSI patch v0.5] <...>) ready.
QUIT
221 Goodbye.
Connection closed by foreign host.
```

Shown above is again the instruction sequence, if everything works as it should. Again, if you see a message about a missing shared library, or the wrong version, the site administrator needs to set up the LD_LIBRARY_PATH correctly.

7.1.3 Verify that you can run simple jobs

Once you verified that you are accepted by the remote site, you may want to verify that you can actually run simple jobs. These tests are two-parts. First, you want to check that you can run a simple fork job, which returns fairly quickly. Again, please replace the “<host>” string with the fully-qualified name of the remote gatekeeper host:

```
globus-job-run <host>/jobmanager-fork -l /bin/date
Fri Sep 12 15:52:59 CDT 2003
```

The above instructions run a simple job in interactive mode. Usually, the remote site has at least one other jobmanager installed, which points to a job scheduling system, e.g. Condor, PBS or LSF. The GVDS will use these jobmanager to run compute jobs. Thus, you will want to verify that you can actually submit jobs to these systems.

The difference is the suffix on the jobmanager – please replace the “<sched>” string with the appropriate remote scheduling system. The remote site will have published information which is the correct jobmanager contact string to use. Please check with the remote site for any required additional arguments like a project name for accounting, or a queue name. Please refer to the Globus documentation on how to set these additional arguments, e.g. `globus-job-run --help`.

```
globus-job-run <host>/jobmanager-<sched> -l /bin/hostname -f
worker.node.in.pool
```

The `-f` option is only necessary for Linux pools. Please note that the execution of this command usually takes five to ten minutes to execute, so patience is required.

If your jobs fails for weird reasons, the remote pool usually contains in your account a file that has a prefix of `gram`, a suffix of `.log`, and contains a process id among other things. Such a file is generated for each Globus job. It will be automatically removed for successful jobs, but remains for failed jobs. This file provides valuable insights for anybody trying to debug jobs at the Globus level. Frequently, if Condor-G experiences problems, developers may ask for this file, too. Its information is admittedly arcane, and does not always illuminate the reason for a failure.

7.1.4 Verify that you can transfer files

In order to private data to your computation, the GVDS stages files between computation nodes. Thus, it is mandatory to check, if you can access files on a remote system.

```
date > test.out
globus-url-copy -vb file:///~/bin/pwd`/test.out gsiftp://<host>/tmp/test.out
                29 bytes          0.03 KB/sec avg          0.03 KB/sec inst
echo $?
0
```

The above creates a small test file `test.out`. In the next step, this test file is being transferred into the temporary space of the remote gridftp server. Please replace the string “<host>” with the fully-qualified hostname of the remote site’s gridftp server. In many cases, it is identical to the gatekeeper, but this is not a requirement. Also note the use of back-quotes to obtain the correct current working directory. The `-vb` option prints additional information about the progress of the transfer. You must check the exit code (`$?`) of the run to verify that it indeed ran successfully.

If the transfer fails, it is often helpful to run with the additional option `-dbg`, and request help with that output. A common problem is the firewalling of the remote site, or your own submit host’s firewall. Section 8.1 contains a few details for an automated test.

7.1.5 Verify that you can run local Condor jobs

In the next step, we will try to verify your submit host environment. This is only necessary to verify once. For local submission, the Condor scheduling system is being used. Condor contains an extension Condor-G, which we will test in the following section.

To verify that your local Condor installation is capable of execution jobs correctly, please create a textual file with the following content:

```
universe = scheduler
executable = /bin/date
output = try.out.txt
error = try.err.txt
log = try.log.txt
copy_to_spool = false
transfer_executable = false
```

```
notification = NEVER
queue
```

In the next step, submit this file to the Condor system. You only type the first line. The next three lines show the expected response from the Condor system, where “<ddd>” is an integer number:

```
condor_submit try.sub

Submitting job(s).
Logging submit event(s).
1 job(s) submitted to cluster <ddd>.
```

You can now occasionally poll the Condor queue to check on your job. Since this jobs will execute almost instantaneous, you won’t see it in the Condor queue. It may be easier, though, to check the job log file `try.log.txt`. The expected output file looks some *similar* to the following:

```
000 (ddd.000.000) 09/12 16:12:18 Job submitted from host: <128.135.11.143:50576>
...
001 (ddd.000.000) 09/12 16:12:18 Job executing on host: <128.135.11.143:50576>
...
005 (ddd.000.000) 09/12 16:12:18 Job terminated.
    (1) Normal termination (return value 0)
        Usr 0 00:00:00, Sys 0 00:00:00 - Run Remote Usage
        Usr 0 00:00:00, Sys 0 00:00:00 - Run Local Usage
        Usr 0 00:00:00, Sys 0 00:00:00 - Total Remote Usage
        Usr 0 00:00:00, Sys 0 00:00:00 - Total Local Usage
    0 - Run Bytes Sent By Job
    0 - Run Bytes Received By Job
    0 - Total Bytes Sent By Job
    0 - Total Bytes Received By Job
...
```

Condor reports a number of things in the status tags, where tag 000 means that a job was being submitted, e.g. recognized by the Condor system. Tags 001 means that the job was actually started. Tag 005 is being generated, if the job finished. Special attention should be put to the return value reported for *local* jobs. If you were getting an unsuccessful run, you will not see the return value of zero for local jobs.

The output file `try.out.txt` or the error file `try.err.txt`, which were connected to `stdout` and `stderr` respectively, may provide insights on the reason of the failure. In the above example, the `try.out.txt` should contain a valid output from the Unix `date` command.

7.1.6 Verify that you can run Condor-G jobs

The Condor-G jobs facility is similar to plain Condor. The difference is, though, that it runs jobs in the Grid using Globus. In order to test these jobs, you need to modify the previous submit file slightly, and put it into a file `try2.sub`. Again, you need to replace “<host>” with the gatekeeper hostname, and “<sched>” with the correct scheduler name:

```
universe = globus
executable = /bin/hostname
globusscheduler = <host>/jobmanager-<sched>
remote_initialdir = /tmp
output = try2.out.txt
error = try2.err.txt
log = try2.log.txt
copy_to_spool = false
transfer_executable = false
notification = NEVER
queue
```

If the remote site requires special projects or queues, you may need to employ RSL syntax to add them. If you don't know what this is, don't sweat it. Otherwise, if the remote site requires a projects, use the (project=<name>) string. If a queue name is required, use the (queue=<queue>) string. You will need to add the globusrsl configuration option into your submit file on a line above the single word "queue":

```
# this is an example, don't use as such
globusrsl = (project=XYZ) (queue=ABC)
```

In the next step, this file is being submitted to Condor, which will in turn process it using Condor-G. The first line again shows the input you need to type, and the next three lines the expected output:

```
condor_submit try2.sub

Submitting job(s).
Logging submit event(s).
1 job(s) submitted to cluster <ddd>.
```

The above line shows that the job got the job id "<ddd>", which is an integer number. This number is used to check on the progress of the jobs, which is, for Globus jobs, easier than checking the log file. To check on job status, the condor_q command is being utilized:

```
condor_q <ddd>

-- Submitter: hamachi.cs.uchicago.edu : <128.135.11.143:50576> : hamachi.cs.uchicago.edu
ID      OWNER      SUBMITTED  RUN_TIME ST PRI  SIZE CMD
ddd.0   voeckler      9/12 16:32  0+00:00:00 I  0   0.0  date
```

The plain command shows the job like any other Condor job. The jobs will most of its life-time shown as being "idle". This is the *local* view that the Condor on the submit host has. Since Grid jobs are special, the option -globus allows to obtain better information about the running job. This information contains the status that the job has on the remote gatekeeper:

```
condor_q -globus <ddd>

-- Submitter: hamachi.cs.uchicago.edu : <128.135.11.143:50576> : hamachi.cs.uchicago.edu
ID      OWNER      STATUS  MANAGER  HOST              EXECUTABLE
ddd.0   voeckler      UNSUBMITTED condor   <remote gatekeeper> /bin/date

condor_q -globus <ddd>

-- Submitter: hamachi.cs.uchicago.edu : <128.135.11.143:50576> : hamachi.cs.uchicago.edu
ID      OWNER      STATUS  MANAGER  HOST              EXECUTABLE
ddd.0   voeckler      PENDING condor   <remote gatekeeper> /bin/date
```

The first line shows the output from a job that was submitted, but has not made it into the remote scheduling system. The next line shows that the remote jobs was accepted into the remote scheduling system, but has not started yet. There exist various other remote job states that may be of interest. For grid debugging, the remote jobs state is way more interesting to a developer than the local Condor state.

Furthermore, for each user, all his or her grid jobs report into a log file just for grid jobs. If the submit host is set up as shown earlier, and extensive log can be found in the /tmp directory, named GridmanagerLog and containing the user account name as suffix. Developers are very interested in this log file for any kind of grid problem in connection with Condor-G.

After a while, approximately five to ten minutes, the job should be done – it may take longer on busy systems. If you check the job log file try2.log.txt, the information will be similar, but longer:

```
000 (ddd.000.000) 09/12 16:32:51 Job submitted from host: <128.135.11.143:50576>
...
017 (ddd.000.000) 09/12 16:33:05 Job submitted to Globus
    RM-Contact: e.cs.uchicago.edu/jobmanager-condor
    JM-Contact: https://e.cs.uchicago.edu:49153/19305/1063402370/
    Can-Restart-JM: 1
...
```

```

001 (ddd.000.000) 09/12 16:38:14 Job executing on host: e.cs.uchicago.edu
...
005 (ddd.000.000) 09/12 16:38:24 Job terminated.
    (1) Normal termination (return value 0)
        Usr 0 00:00:00, Sys 0 00:00:00 - Run Remote Usage
        Usr 0 00:00:00, Sys 0 00:00:00 - Run Local Usage
        Usr 0 00:00:00, Sys 0 00:00:00 - Total Remote Usage
        Usr 0 00:00:00, Sys 0 00:00:00 - Total Local Usage
    0 - Run Bytes Sent By Job
    0 - Run Bytes Received By Job
    0 - Total Bytes Sent By Job
    0 - Total Bytes Received By Job
...

```

There are a few subtle differences worthy of your attention:

1. There is an additional tag 017, which is specific to Globus jobs. This tag is being generated once the Condor-G subsystem delivered the job to the remote gatekeeper.
2. Although the 005 tag contains a return value, the value reported by Condor-G is always 0. This is effectively Globus' fault by lacking the appropriate reporting facility. The lacking return code is also the reason we highly recommend using some kind of grid launcher (kickstart, gridsh) to run remote jobs. The grid launcher will provide the submit site with the remote result code.

There are now a lot more reasons possible for your job failing. Some errors are simple, e.g. if the job submission returns with the message that the proxy filename could not be determined, your grid proxy certificate has expired.

With the long option “-l” to `condor_q`, you can check in more detail, what went wrong with a failed job, if you provide the ID that Condor told you about the job:

```
condor_q -l <jobid> | sort
```

The result is a wealth of information. The *HoldReason* usually contains the Globus error, if Globus was at fault for holding a job. For instance, if the error reads that stdout or stderr could not be opened, most likely a firewall issue prohibits Globus from transferring GASS files:

```
condor_q -l | sort | grep ^Hold
```

The above returns all reasons a job was put on hold for all jobs. Provide a job id to obtain information about a specific job.

7.1.7 Verify that you can run DAGMan in its full beauty

Ok, this part needs to be written, but *usually* there are not many problems with DAGMan.

7.2 If nothing else helps

The GVDS maintains several facilities to help out users, and to dispatch user problems to the fabric's help facilities. If you think you discovered a bug in GVDS, please use the bugzilla to report the bug:

<http://bugzilla.globus.org/chimera/>

If you are not quite sure about the nature of a true bug in the virtual data system, or if it is a bug at all, you may use the chimera-support@griphyn.org mailing list. For general discussion, you may want to post request to chimera-discuss@griphyn.org. Currently, these lists are open until further notice. Should they become closed, you must subscribe, as detailed below.

When submitting bug reports via email, it is a good idea to collect and send along information about the various systems that are involved. This includes the version of VDT (`vdt-version`), the version of VDS (`vds-version`), the version of the offending command (`<cmd> -V` or `<cmd> --version`), the operating system and version (`uname -a`), the remote operating system and version. For bugs related to the actual execution of

jobs, please also include the pertaining parts from the grid manager log file on the submit host, and, if applicable, the remote sites jobmanager log file.

Finally, if you would like announcements about new releases, urgent bug-fixes and user polls, you can subscribe to [chimera-announce](#) by sending a message with no subject, and the `*only*` body

subscribe chimera-announce

from your regular email account to `majordomo@griphyn.org`.

8 Test Scripts

There are a couple of test scripts available. This section will deal with some integration tests that the user should run to verify that VDS could indeed be run successfully.

8.1 Verifying Your Foundation

Before you try to run any VDS software, it is strongly recommended that you verify your foundation software. The foundation software consists of Globus, Condor-G and Java.

The picture to the right is an approximation of the interactions between the foundation layers of software. The directory `$VDS_HOME/test/test0` contains the verification software for the foundation layers.

In order to run the verification script `test.pl`, you will need to have set up your Globus, Condor and Java. It is assumed that you run the verification on the submit host. Thus, network access is mandatory. Also, remember to initialize your user certificate before running the test script.

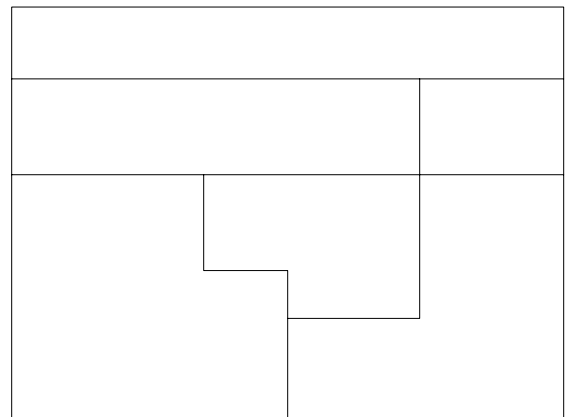
The test script is a Perl script. It is assumed that at least version 5.005 is installed, though version 5.6.1 is recommended. If you have any problems with missing modules, you might want to refer to the contrib directory for the missing module `File::Temp`. The test script is on purpose kept compatible to older versions of Perl, thus not featuring many convenience modules.

To find out about the options to steer the test script, run it with the `--help` option. By default, the test assumes that the machine you run the test upon has a gatekeeper with a regular jobmanager installed, and that the host also features a grid-enabled ftp server. If you want to contact other hosts, please look into the options `--contact` and `--gridftp`. You can repeat these options to test multiple destinations. Personal gatekeepers are also permissible.

Most tests are fairly thorough. There are three report levels. “OK” means that the test ran all right. “Warning” constitutes an unsure situation. The test is sure that something is not as it should be, but it will try to continue testing, though it may fail later on. The “error” message leads to a termination of the tests. Usually, some diagnostic messages should help you to locate the problem with the software or configuration underneath.

Here are your results:

```
test suite globus.....: OK
test suite gridftp.....: OK
test suite condor.....: OK
test suite javavm.....: OK
test suite chimera.....: OK
```



After running the test script, the final output should look like the output above. Please note that in this release, some tests are not very elaborate, and contain FIXME sections to remind the author of tests to be written. You can safely ignore the FIXME sections.

8.2 Running VDS samples

The next samples run the kanonical executable for grids (*keg*) on remote pools. The tests in the directories *test1* through *test4* are similarly built. Both times, a script *test.sh* needs to be run, which requires a similar set of command line arguments. The setup of the *pool.config* file in the parent directory of the tests needs to be set up correctly to match your environment. This section describes the common setup, while the subsections describe the tests themselves. The upcoming sections will also deal with the *tc.data* setup.

The *keg* binary is provided as statically linked version for Linux in the *bin* directory for your convenience. In order to run the tests, you need to copy the *keg* binary into the pool, and make it available to each worker node. Please run it as follows on one node of the pool to verify that indeed it is runnable in your pool. Mine yields the following results – yours should look similar, but not identical.

```
$ bin/keg -o /dev/fd/1

Timestamp Today: 20020828T132047-05:00 (1030558847.754;0.004)
Applicationname: keg @ hamachi.cs.uchicago.edu (128.135.11.143)
Current Workdir: /home/voeckler
Systemenvironm.: i686-Linux 2.4.10-4GB-SMP
Processor Info.: 2 x Pentium II (Deschutes) @ 400.916
Output Filename: /dev/fd/1
```

The purpose of the *keg* binary is to copy all its input files onto all its output files while appending each output file with the above information stamp. Thus, *keg* can be used as a stand-in for real applications, and helps to track the flow through a DAG. Please refer to its manual page for more information.

While you are at it, please remember that you must pre-stage each application. The *globus-url-copy* and *transfer* applications are used by the automatically generated transfer nodes. Thus, they are expected to exist at each remote pool, and likewise visible to each worker node. While you are still logged into the remote pool, you might as well want to check, if the binaries work for you. First, you have to have a valid grid user proxy certificate – if not, run *grid-proxy-init*. Then, try to obtain a file from some grid ftp service that you have access to. Please note, even though we talk about grid ftp, the URL schema is *gsiftp*.

Since you are in the remote pool anyway, create a directory *\$HOME/vlldemo*. The test assume that this directory is shared and visible to all worker nodes. The tests will stage into this directory, compare with the *pool.config* file in the next section.

In the next step, you need to adjust the *pool.config* file in the parent directory of the *test1* through *test4* directory. There is a sample configuration for my own tests *pool.config.rls*. While you should not blindly copy it into *pool.config*, it may give you clues how to set up your own configuration file.

The pool file must contain the pool handles that you intend to use. It is recommend to supply information pertaining to the *local* pool handle, if your submit host is configured to work as a personal Condor. The format of the *pool.config* file was described previously in section xx.x. A typical pool file might look like this, assuming your run pool has a handle *ufl*.

uofc	vanilla	your.gk.host/jm-condor	gsiftp://your.gfp.host/home/vlldemo	/vlldemo	/vds/bin/kickstart	rls://your.rls.server
uofc	globus	your.gk.host/jm-condor	gsiftp://your.gfp.host/home/vlldemo	/vlldemo	null	null
uofc	transfer	your.gk.host/jm-fork	gsiftp://your.gfp.host/home/vlldemo	/vlldemo	null	null
local	vanilla	submit.host/jm-condor	gsiftp://submit.host/vlldemo	/vlldemo	/to/kickstart	null
local	globus	submit.host/jm-condor	gsiftp://submit.host/vlldemo	/vlldemo	null	null
local	transfer	submit.host/jobmanager	gsiftp://submit.host/vlldemo	/vlldemo	null	null

Please note that the “globus” universe is being deprecated, but still may be required for older versions of the VDS system. The “globus” universe look like the “vanilla” universe. The “transfer” universe should point to a fork jobmanager, if the pool is fire-walled. The test scripts of the following sections have a uniform invocation behavior. The script takes at least two mandatory parameters, and may be augmented with further parameters:

- The `--rls` parameter takes the name of a replica catalog, e.g. `rls://shoveled.mcs.anl.gov` in ANL. You need to contact the administrators to create a replica location service for your perusal. This is a mandatory parameter.
- The `--run` parameter takes a pool handle as argument. The pool will be used to run the transformation at. This is a mandatory parameter.
- The `--src` parameter takes a pool handle as argument. The pool handle usually describes your own submit host. The source will solely be used to create the initial file `data.in`, for instance, and to stage the file to the execution pool. The default for the source handle is `local`.
- The `--dst` pool handle argument describes the final resting place for data. The data file `data.out` will be staged out into this pool, and registered in the replica catalog for this pool. The default for the destination pool is `local`.
- The `--stop-after-cplan` argument is intended for debugging purposes. It stops the script after the concrete planner generated the necessary information for Condor DAGMan, but before submitting the DAG to Condor. It is highly recommended to use this option, and look closely at the generated files or error message before blindly submitting any job.

The handles for source, execution and destination pool may at your choice be all the same, or differ in any manner. It is recommended, however, to use the submit host as destination pool, and set up the pool handle `local` accordingly, if applicable.

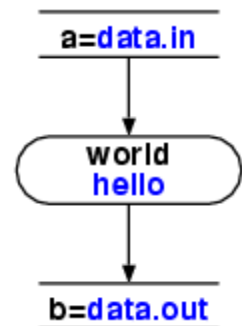
Before running any of the further tests, it is highly recommended to clean out your portion of the replica catalog.

8.2.1 Hello World

The hello world example is a very simple one job, one input file, one output file simulation. The test can be found in the `$VDS_HOME/test/test1` directory. The hello world example runs the `keg` binary to convert the input data into the output data.

```
TR @USER@::world:1.0( input a, output b )
{
    argument = "-a hello";
    argument = " -i "${input:a}";
    argument = " -o "${output:b}";
}

DV @USER@::hello:1.0->@USER@::world:1.0,1.0(
    a=@{input:"@USER@.data.in"},
    b=@{output:"@USER@.data.out"}
);
```



Please refer to appendix 9 for a description of VDLt. The token “@user@” will be replaced by the user of the Unix account who runs the test scripts. This is to avoid clashes between different users running the same set of tests, including clashes in the replica catalog.

The transformation in this case is called *world*, or rather `<user>::world:1.0`. It takes an input file argument *a* and an output file argument *b* to run. Within, the command line of the application linked to the transformation by the transformation catalog in the *tc.data* file, will be invoked with a string argument, the input filename and the output filename. The transformation data is shown in black in the dataflow chart.

The derivation *hello*, fully qualified `<user>::hello:1.0`, instantiates the *world* transformation. It passes the file `<user>.data.in` for argument *a* and `<user>.data.out` for argument *b*. The derivation is shown in blue (gray) in above diagram.

The filenames refer to logical files. The replica catalog provides the mapping between these logical filenames and their physical counterparts. The *test.sh* script populates the replica catalog with the logical file `<user>.data.in`, and removes any occurrences of occurrences `<user>.data.out`.

Before running the hello world test from its test script *test.sh*, it is strongly recommended that you open the supplied *tc.data.in* in an editor, and modify it to suit your own configuration. Its syntax is described elsewhere in this document. Make sure that you have an entry for the *world* transformation and the pool you chose to run in. It should map to the pre-staged *keg* executable. Also make sure that the pool has entries for *globus-url-copy*, and that it is available in the run pool. Finally, make sure that the submit site is the *local* pool handle, has an entry for *RLS_Client*, and that it maps to the *rls-client* wrapper on the submit host.

A sample minimal *tc.data.in* is shown below, with the environment variable settings abbreviated:

```
uofc  @USER@_world_1.0  /home/voeckler/bin/keg  null
uofc  globus-url-copy  /vdt/bin/globus-url-copy  GLOBUS_LOCATION=...;LD_LIBRARY_PATH=...
local  RLS_Client      /path/to/sh-vds/bin/rls-client  VDS_HOME=...;JAVA_HOME=...;CLASSPATH=...
```

The output file `<user>.data.out` will be displayed after a successful run. If the host *hamachi* is the *local* pool, and is also the destination pool, you might be able to see something similar to the following result. The input file `<user>.data.in` is repeated in the result. It is just a concatenation of the current *date* output with the output from *hostname -f*.

```
voeckler@hamachi:~/vdl-demo> head voeckler.data.*
==> voeckler.data.in <==
Mon Oct 27 14:08:29 CST 2003      hamachi.cs.uchicago.edu
==> voeckler.data.out <==
--- start voeckler.data.in ----
  Mon Oct 27 14:08:29 CST 2003  hamachi.cs.uchicago.edu
--- final voeckler.data.in ----
Timestamp Today: 20031027T141115-06:00 (1067285475.599;0.044)
Applicationname: hello @ 128.135.11.xx (xx.cs.uchicago.edu)
Current Workdir: /home/voeckler/vdl-demo
Systemenvironm.: i586-Linux 2.4.yy
Processor Info.: 1 x AMD-K6(tm) 3D processor @ 400.921
Output Filename: voeckler.data.out
Input Filenames: voeckler.data.in
```

8.2.2 A More Complex Example – A Black Diamond

The black diamond is a more complex version of the diamond DAG. The black diamond is the final litmus test for ensuring that you will be able to run your own VDS examples successfully. The black diamond test can be found in the `$VDS_HOME/test/test3` directory.

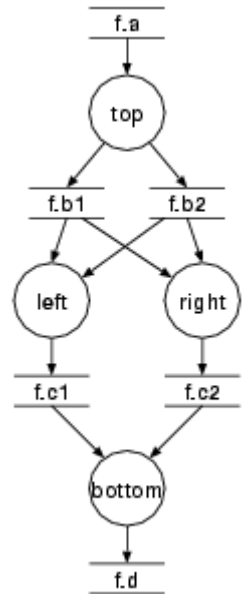
Each transformation in the black diamond will run the *keg* binary. The dataflow chart to the right only shows the derivation chain:

- The derivation *top* consumes one file $\langle user \rangle.f.a$, and produces two files $\langle user \rangle.f.b1$ and $\langle user \rangle.f.b2$.
- The derivations *left* and *right* both consume $\langle user \rangle.f.b1$ and $\langle user \rangle.f.b2$. The output files $\langle user \rangle.f.c1$ and $\langle user \rangle.f.c2$ are produced by these derivations respectively. As with the regular diamond, both *left* and *right* instantiate the same transformation.
- The derivation *bottom* takes the previously produced files, and combines them into the final result file $\langle user \rangle.f.d$.

Again, when the *test.sh* script is run, the replica catalog is emptied of any produced file. The local file $\$HOME/vdldemo/\$LOGNAME.f.a$ will be produced and registered with the RC for the source pool (this is the reason why we recommend to use *local* as source pool). All other product files are removed from the replica catalog. In order to create a location entry in the RC for the destination pool, a *dummy* file is entered.

Before running the black diamond test from its test script *test.sh*, it is strongly recommended that you open the supplied *tc.data* file in an editor, and modify them to suit your own configuration. Its syntax is described elsewhere in this document. Please make sure, as before, that the transformations *analyze*, *findrange* and *generate* map to the *keg* binary for your run pool, that it contains a mapping for *globus-url-copy* for the run pool, and a *RLS_Client* for the *local* pool.

The results from a successful run should display plenty of *keg* information. It is too long to repeat here, but it should enable you to trace the hosts that the jobs ran on.



9 Appendix I: Glossary of Terms

Virtual data	Denotes data objects whose method of production (or reproduction) is accurately known and represented in a metadata repository (specifically, this case, the virtual data catalog).
Virtual data model	(VDM) The schema of the virtual data catalog that represents the provenance of virtual data objects.
Virtual data language	(VDL) A language that describes how a virtual data object is produced.
VDLt	A textual representation of the VDL, intended to be human readable and producible.
VDLx	An XML representation of the VDL, intended for application-to-application interchange.
VDC	Virtual data catalog. In this Beta release, the VDC consists of an ordinary file that stores a set of XML elements.
VDS	The virtual data system, including Chimera and Pegasus.
Chimera	The system that includes the VDC and associated application tools.
Pegasus	The Planning and execution system, that instantiates the abstract DAG and produces a concrete DAG, which it can send to DAGMan for execution.
Logical file (LFN)	A set of data contained in a file and given a name (the “logical file name”) that is independent of the files location (i.e., on which site in the grid the file resides)
CLI	command-line interface: An application that typically runs in a shell window.
Abstract dag	An abstract directed acyclic graph (aDAG) with logical names of both, executables and files
Concrete dag	A concrete condor style DAG which can be submitted to the Condor DAGMan, it contains specific executable and input/output file locations.
DAG	A directed acyclic graph that expresses the order in which a sequence of jobs is to run, and the dependencies between the jobs
DAX	An abstract DAG expressed in XML
PTC	The Provenance Tracking Catalog, which keeps invocation records
RSL	The Globus resource specification language, not to be confused with RLS
RLS	The Replica Location Service, describes the set of RLI and one or more LRCs
RLI	The Replica Location Index, an index server that knows which LRCs have an LFN. Note, false positives must be checked with the LRCs that are referred
LRC	The Local Replica Catalog contains knowledge for the LFN mapping.
Transformation	Any executable or script
Derivation	Recipe for or record of an instantiation of a transformation with specific arguments
Condor	University of Wisconsin High Throughput Job Scheduler
Condor-G	An extension to Condor that supports Globus
DAGMan	Condor's directed acyclic graph manager

Abstract planner	The Chimera decision making engine which generates abstract DAGs in XML
Concrete planner	The Pegasus planner, which takes a DAX and generates DAGs.
Storage element	Gridftp-enabled host that shares its file system(s) with one or more Compute Element.
Compute element	Grid enabled cluster of compute hosts all of which share access to the file system of a Storage Element.
Catalog host	The catalog host (CH) has access to the Virtual Data Catalog. The abstract planning takes place on the catalog host.
Submit host	The submit host executes the constructed DAG. Usually, it will be identical to the catalog host.
Properties	The configuration parameters for the VDS
Hint	Hints, provided in the VDC and thus copied into the DAX, enable the VDS system to override default behaviors. For example, the pfnHint, if specified, overrides the data in the transformation catalog.
Profile	A profile allows to pass configuration information to later stages in the planning process in a system-independent manner.
Wrapper	The scripts provided in the bin directory of the release, which call the underlying java classes.
Transfer nodes	The nodes which are added to get the data to the execution pool from the locations as returned from the replica catalog.
Replica nodes	The nodes which are added to transfer the materialized data from the execution pool to the output pool, and register the materialized data in the Replica Catalog.

10 Appendix II: Primer to the VDLt textual input language

Please refer to the human-readable PBNF (“people-BNF”) for VDLt. The definitive specification for VDL, however, is the VDLx XML schema. We expect that TR are usually hand-crafted by a human, and thus are likely to use VDLt. The DV on the other hand are expected to be generated in some way. Any generator can as easily generate the preferred XML format as it could generate VDLt. Really! Do not generate VDLt for DVs.

Please refer to <http://www.griphyn.org/workspace/VDS/> and the PBNF for a more formal specification of the input language. There is also another tutorial found at <http://www.griphyn.org/workspace/VDS/langref/>

10.1 Overview

```
"TR" fqid "(" farglist ")" "{" bodylist "}"
"DV" fqid "->" fqm "(" aarglist ")" ";"
```

An `fqid` is a fully qualified definition identifier, has an optional namespace, a mandatory name, and an optional version. The namespace, if it exists, is separated with two colons from the name, and the name with one colon from the version. The `fqm` is a match. At its core it has the same syntax as an `fqid`, but the version information is a comma-separated min,max version list instead.

Please note that a DV does not have a body. The body of a TR is optional.

10.2 The Transformation TR

The `farglist`, list of formal arguments, names each argument with a "type", possibly a container signifier, and an optional default value. The type can be of `none`, `input`, `output`, `inout`, and may be abbreviated as (nothing for `none`), `in`, `out`, `io`. The argument is scalar, unless the argument name is followed by a `[]`. The optional default value is introduced by a `"="` followed by any number of legal text and LFN values. Multiple arguments in the formal argument list are separated by a comma. Currently, the position of an argument is of no relevance, as they are bound by name (like Python). However, future versions may work with transformation signatures, and thus will bind arguments to a position. A few examples for the formal argument list show what is permissible in a TR:

```
TR a (in a, out b)
TR b (p = "0.5", out f = @{out:"lfn1"} )
TR c( in a[] = [ @{in:"lfn1"}, @{in:"lfn2"} ], out b )
```

The TR body is a list of argument or profile 'lines', each terminated by a semicolon. The ordering of the arguments will be kept, while the ordering of profiles are deemed arbitrary:

```
"argument" [ id ] "=" { text | use } ";"
"profile" ns."name" "=" { text | use } ";"
```

The argument `id` is optional, unless you use one of the reserved `stdin`, `stdout` or `stderr`. Within the TR body, only textual or references to bound variables may be used. Since the "type" is established in the `farglist`, it may be omitted when using a variable within the TR body:

```
argument = "-f " ${a};
argument stdin = ${in:b};
argument = ${ "[" ":" "," ":" "]" | c };
profile hints.doesnotexist = " -f " ${none:b} ${none:a} " 0.523";
```

LFNs are **not** allowed **within** a TR body. For a TR, they are only allowed for defaults in the formal arguments list. In the examples above, the first argument consists of a string minus eff space, and the current value of variable "a", which must have been declared in the formal argument list. It will be replaced with the current value of "a". The second argument connects the LFN from \$b with `stdin`. The third example is the rendering of a list with a prefix string *open bracket*, an item separator *comma*, and a list suffix *close bracket*, of the list contained in variable `c`. The profile example concatenates the verbatim strings with the current value of variables `a` and `b`. In this sense, it works some type-casting by assigning a `none` type, and treating the files names as string constants.

10.3 The Derivation DV

In a DV, you associate each value of a given formal argument with a value. Only verbatim text and LFNs are allowed in a DV actual argument list. Each formal argument must have an actual argument associated with it, unless the formal argument provided a default value.

```
DV me->a( b=@{out:"lfn2"}, a=@{in:"lfn1"} );
DV mo->b( );
DV ma->c( a = [], b = @{out:"lfn3"} );
```

The first DV associates a value with each formal argument `a` and `b` of TR `a`. Note how the arguments are bound by name, not position. Also note that currently, the names for TR and DV, and the argument names are in disjunct scopes.

The second DV does not provide any value to the TR `b` that it calls. It uses the default values from the TR. The example has more the benefit of showing what is do-able than being sensible.

The third DV passes an empty list to overwrite the default list. It must pass something for the "b" value to avoid an error.

10.4 The Other Elements

The previous examples already familiarized you with the additional elements that construct values:

Verbatim text is usually enclosed in quotes. The quote itself can be backslash escaped in well-known C-fashion. The backslash itself thus must be backslash escaped. Verbatim text is allowed in all places where values are required. Specifically, it is allowed within a TR, in the default values for a formal argument, or as supplied value for an actual argument to a type `none` variable. VDL, as of this writing, does not distinguish between numerical and string values.

Logical Filenames can have two valid formats, as shown below. Both formats start by an at symbol, followed by curly braces. Within, the file type is specified, which is one of `input`, `output` or `inout`. The abbreviations are permissible. The second part is a quoted string, like the verbatim text. It represents the logical filename value that will be used for look-ups in the replica catalog (with the restrictions and extensions detailed in section x).

```
"@{" filetype ":" qstring "}"
"@{" filetype ":" qstring ":" qstring "}"
"@{" filetype ":" qstring [ ":" qstring ] "|" rt "}"
```

The first format shown is used for regular logical filenames that need to be staged to and from a DAG. Regular LFNs get entered into the replica catalog.

The second format will be used for temporary names. It is not supported in the initial release. A temporary name has a third section, a quoted string like the verbatim text, which provides a hint in the selection of a temporary name. Temporary filenames may be assigned by the concrete planner. Temporary files are used to glue two derivations together. Although they *may* require to be staged in and out for a job, they do not participate in the stage-in and out of results from a DAG. Temporary files may go away after all dependent DVs were executed. Temporary filenames are **not** entered into the replica catalog.

Logical filenames are permissible as default values in formal argument lists, and as values for actual arguments being passed by a DV. They are not permitted within a TR body.

A finer-grained control is possible by providing the argument “r” and/or “t” after a vertical bar. If the “r” argument is present in a logical filename, the file will be registered in the replica manager. If the “t” is present, the file will be staged, except for inter-pool transfers. Providing the temporary file pattern string is equivalent to specifying both, “r” and “t” for backward compatibility. Thus, two equivalence classes exist:

```
"@{" filetype ":" qstring "}" == "@{" filetype ":" qstring "|" rt "}"
"@{" filetype ":" qstring ":" qstring "}" == "@{" filetype ":" qstring ":" qstring "|" "}"
```

VDLx users should note that the `rt` flags are converted into their inverse meaning in VDLx. If you generate VDLx, you should always specify exactly what you mean using the *dontRegister* and *dontTransfer* boolean attributes, and treat the *temporaryHint* attribute as just a string.

Bound variables refer to a parameter in the formal argument list. Bound variables may only be used within the transformation body. They are not permitted in any argument lists.

```
${a}
${none:c1}
${" "|lst}
${"[" ":" "," ":" "]" |output:olst}
```

The first example refers to the bound variable `a`. Please note the introductory dollar sign, and the enclosing curly braces around the values. Since the type of `a` is declared in the formal argument list, it may be omitted. The second example uses a scalar variable that is verbatim text.

The third example uses rendering for a list value. The list `lst` will be rendered by putting spaces between each element. The fourth example extends the rendering to enclose the list `lst` in brackets, and separate each element with a comma from one another.

10.5 A Diamond Example

People familiar with Condor DAGMan will know about the diamond DAG. This section will show which VDL input specification can create a diamond DAG.

```

TR generate( output a )
{
    argument stdout = ${output:a};
    profile hints.pfnHint = "generator.exe";
}

TR findrange( output b, input a, none p="0.0" )
{
    argument arg = "-i "${none:p};
    argument stdin = ${input:a};
    argument stdout = ${output:b};
    profile hints.pfnHint = "findrange.exe";
}

TR analyze( input a[], output c )
{
    argument files = ${a};
    argument stdout = ${output:c};
    profile hints.pfnHint = "analyze.exe";
}

DV left->generate( a=@{output:"f.a"} );
DV top->findrange( b=@{output:"f.b"}, a=@{input:"f.a"}, p="0.5" );
DV bottom->findrange( b=@{output:"f.c"}, a=@{input:"f.a"}, p="1.0" );
DV right->analyze( a=[ @{input:"f.b"}, @{input:"f.c"} ],
c=@{output:"f.d"} );

```

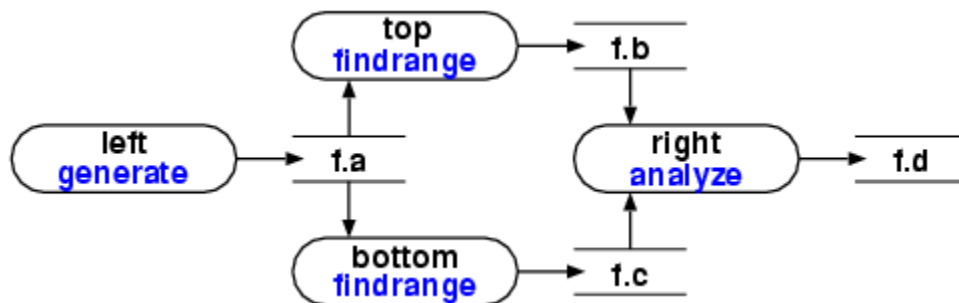


Figure 7: A diamond DAG.

The above example first defines three transformations: “generate”, “findrange” and “analyze.” The derivations “top”, “left”, “right” and “bottom” invoke these invocations. The derivations were named in this example according to their position in the diamond DAG. Also note that the transformation “findrange” is invoked twice, with different parameters.

The derivations are linked through their data dependencies. The “left” derivation only invokes the generator. Looking into the transformation, you will notice that only one file “f.a” is produced, and no other arguments are used. The “top” and “bottom” derivations both take the produced file, and apply some mathematics of their own, steered by the value for parameter “p”. They produce files “f.b” and “f.c” respectively. Finally, the “right” derivation combines the previous results in “f.b” and “f.c”, and produces the summary in “f.d”.

In order to generate the diamond for DAGMan, you request the DAX for the production file “f.d”, or you request the derivation “right”. Either way, the abstract planner will produce the complete build for the diamond as a DAX. In the next step, the concrete planner will convert this build-style recipe into a make-style recipe. This means, the concrete planner will check the replica catalog for existing files, and thus for computational steps it can avoid. If your replica catalog is pristine with regards to files “f.a” through “f.d”, the concrete planner will produce the diamond DAG for DAGMan, including steps for the stage-out of files, and registering them with the replica catalog.

The above example instantiates just one diamond DAG. One can use the same three transformations to instantiate any number of derivations, and thus diamonds. If multiple diamonds are to be constructed, and if the intermediary files “f.a” through “f.c” are transient, the recipe for a diamond might be better kept within a compound transformation.

11 Appendix III: Managing the replica catalogs

This section contains the details on how to access the two replica catalog implementations that VDS currently supports.

11.1 The Globus-2 Replica Catalog

11.1.1 Creating the setup_rc file

This class is used to generate a properties file containing the various options to connect to a Replica Catalog. This file can then be used to override the properties specified in the \$VDS_HOME/etc/properties file.

```
replica-catalog-init -host <host dn> -mdn <manager dn> -pass <password> \
-f <output setupfile>
```

e.g

```
replica-catalog-init --host \
"ldap://smarty.isi.edu:4156/rc=TestReplicaCatalog, ou=isi.edu, o=USC" \
--mdn "cn=Directory Manager, ou=isi.edu ,o=USC" --pass drowssap
```

Note: enclose the hostdn and mdn in quotes to protect it from the shell.

11.1.2 Java Client to access the replica catalog servers

The *replica-catalog* command uses the parameters in the VDS properties file found in the \$VDS_HOME/etc directory to connect to the Replica Catalog. Other options are passed as command line arguments. This command is a Java version of the *globus-replica-catalog* command and is based on the COG kit. The command can be used to create logical collections in the replica catalog, associate filenames and locations with it. It also provides the ability to delete collections and locations.

The user has three options for specifying the logical filenames.

1. Add the filename by using --addFile option.

2. Enter multiple filenames on the command line by using the `--addMulFiles` option.
3. Enter multiple filenames in a text file such as `file.txt` and input it to the command using the `--addFiles` option.

At present there is no one to one mapping of options between the c and the java versions. For a complete reference please refer to the man pages.

Below are some examples of to use the tool.

11.1.3 Creating a collection

```
replica-catalog --collection GriphynData --create --addFiles files.txt
```

Creates a collection named GriphynData , containing logical filenames which are specified in a text file `files.txt`. The collection is created in the directory at the ldap dn specified in the setup file.

```
replica-catalog --collection GriphynData --create --addFile result.F
```

Creates a collection with a single logical filename `result.F` associated with it .

11.1.4 Creating locations

```
replica-catalog --collection GriphynData --location uc --locationURL
gsiftp://condor.cs.uchicago.edu --addLocation --addFiles files000-100.txt

replica-catalog --collection GriphynData --location uw --locationURL
gsiftp://condor.cs.wisc.edu --addLocation --addFiles files100-200.txt

replica-catalog --collection GriphynData --location isi --locationURL
gsiftp://smarty.isi.edu --addLocation --addFiles files200-300.txt
```

After a collection containing logical files has been created, these logical files are associated with locations. The locations specify where a particular logical file resides. The files being associated with the location must be already associated with the collection otherwise an error will be generated. One can alternatively use the `-addFile` option to add a single filename to the location instead of specifying a list of filenames associated with the location in a text file.

11.1.5 Adding filenames to existing location

```
replica-catalog --collection GriphynData --location isi \
--addFiles files100-200.txt

replica-catalog --collection GriphynData --location isi \
--addFile result.F
```

Adding files to an already existing location. Those who are familiar with *globus-replica-catalog*, the adding of filenames follows the `addfiles` option.

11.1.6 Listing the filenames associated with the collection

There are several option how to list the filenames an a G2RC that are associated with a collection:

1. Only the filenames

```
replica-catalog --collection GriphynData -listFilenames
```

2. Only the names of logical files

```
replica-catalog --collection GriphynData --listLogicalFiles
```

3. Names of logical files with their respective attributes

```
replica-catalog --collection GriphynData --listLogicalFiles --listAttributes
```

One can also have logical file objects associated with a particular collection. To display those, use the `--listLogicalFiles` option, and `--listAttributes` to see the attributes associated with the logical files.

11.1.7 Listing filenames with a particular location

```
replica-catalog --collection GriphynData --location isi --listFilenames
```

11.1.8 Listing all the locations associated with collection

```
replica-catalog --collection GriphynData --listLocations
```

11.1.9 Listing all the locations including their attributes

```
replica-catalog --collection GriphynData --listLocations --listAttributes
```

In the replica catalog the location objects and collection objects have attributes associated with them. Giving the `--listAttributes` option displays all the attributes. If this option is not given only the 're' attribute for the object is displayed

11.1.10 Deleting filenames

1. Delete all filenames from a collection

```
replica-catalog --collection GriphynData --deleteAllFile
```

2. Delete all filenames from a location

```
replica-catalog --collection GriphynData --location isi --deleteAllFile
```

3. Deleting filenames specified in a text file files.txt from a collection

```
replica-catalog --collection GriphynData --location isi \
--deleteFiles files.tx
```

4. Deleting filenames specified in a text file from a location

```
replica-catalog --collection GriphynData --location isi \
--deleteFiles files.txt
```

5. Deleting a location from a collection

```
replica-catalog --collection GriphynData --location isi --deleteLocation
```

11.2 The Replica Location Service

11.2.1 Defining the pool attribute to be associated with the PFN

```
globus-rls-cli attribute define "pool" "pfn" "string" rls://<rls-host-name>
```

Please note, this step needs to be done only once on an LRC.

11.2.2 Adding a single LFN to PFN mappings

```
globus-rls-cli create "foo" "gsiftp://tnt.isi.edu/home/people/bar" rls://<rls-host-name>
```

11.2.3 Subsequent additions for the same existing LFN

```
globus-rls-cli add "foo" "gsiftp://smarty.isi.edu/people/gmehta/bar2" rls://<rls-host-name>
```

11.2.4 Adding the pool attribute value for each PFN

```
globus-rls-cli add gsiftp://tnt.isi.edu/home/people/gmehta/bar pool pfn
string <pool-name> rls://<rls-host-name>
```

11.2.5 Adding the entries using bulk mode.

Create a file called “seedrls” with the entries to create lfn-mappings and attribute mappings

```
create foo1 bar1
attribute add bar1 “pool” “pfn” “string” <pool-name>
create foo2 bar2-1
add foo2 bar2-2
add foo2 bar2-3
attribute add pfn2 “pool” “pfn” “string” <pool-name>
create lfn3 pfn3
attribute add pfn3 “pool” “pfn” “string” <pool-name>
```

and so on. Pipe or connect the *stdin* of *globus-rls-cli* to the file you just created:

```
cat “seedrls” | globus-rls-cli rls://<rls-host-name>
```

11.2.6 Querying a LRC

Use the following command to search for a logical filename in an LRC. Please note that wildcards are allowed.

```
globus-rls-cli query lrc lfn “foo” rls://<rls-host-name>
```

11.2.7 Querying a RLI

Use the following command to search for a logical filename in an RLI. Please note that wildcards are permitted.

```
globus-rls-cli query rli lfn “foo” rls://<rls-host-name>
```

11.2.8 Deleting a LFN from a LRC

In order to delete a LFN from a local replica catalog, use the following command syntax.

```
globus-rls-cli delete “foo” “gsiftp://smarty.isi.edu/people/gmehta/bar2” rls://<rls-host-name>
```

11.2.9 Making a LRC update to a RLI

In order to make a LRC update its entries to a particular RLI

```
globus-rsl-admin -a rls://<rli name> rls://<lrc name>
```

11.2.10 Making a LRC force update to a RLI

In order to force a LRC to send updates to a RLI

```
globus-rls-admin -u rls://<lrc name>
```

12 Appendix IV: Miscellaneous Issues

12.1 Namespaces

Transformation and derivation definitions reside in namespaces. Each TR and DV definition is uniquely identified by the triple (namespace, identifier, version number).

Namespaces serve to eliminate identifier clashes between multiple users of a virtual data catalog, or within large projects. A namespace name can be any XML string. The grouping of TR and DV definitions into namespaces is completely up to the user. Users can introduce sub-structure within a namespace through their own naming conventions, e.g.: “cms.cern” or “/atlas/bnl”. Such names are treated as a string - the virtual data system does not currently interpret such structure (although it may in the future).

The <definitions> root element in a VDLx document contains a “vdlns” attribute. This namespace will be used as the default namespace for any transformation or derivation defined or referenced within the <definitions> that does not explicitly specify a namespace. The root element also contains a “version” attribute, which will be taken as the default version for transformations and derivations defined or referenced within the <definitions>.

When a TR is defined, a *namespace attribute* can be specified in the TR definition, or the TR can be placed in the default vdlns namespace defined in the <definitions> element xmlns attribute. When a TR is referenced by a DV in the *uses* attribute, a corresponding *usespace* attribute defines the namespace to search for the TR definition.

12.2 Special properties for future releases

The following properties are targeted to be used with the --head option.

vds.cplanner.work_dir	the working directory. This is used specifically if one is using the single stage in and stage out of files. Also this working directory is relative to the mount point of the execution pool.
vds.cplanner.transfer_pool	this option again is applicable if using the single stage in and stage out. It specifies the pool where the transfer/magic script runs.

Table 11: Properties that are not available in this release

12.3 Tests

- Black box tests, provoking errors.
- Integration tests, also for the user.

13 Known Issues

- The C based client of the replica catalog cannot be used to populate the replica catalog since there is a mismatch between the client and the server code. All data to be used with the VDS system needs to be populated with the java based client of the replica catalog. Any data that has previously been registered with the c client will most probability not work with the VDS
- If the final o/p of the DAG is already available on the output location where it is required, the system will still generate DAGs for transferring it to the execute pool and back to the output pool instead of indicating to the user that the final data is already on the system.

- All the pools listed in the `pool.config` file need to be registered beforehand as locations in the replica catalog. A dummy filename might have to be used in order to create these locations.
- The local Condor-G on the submit Host (SH) needs to be capable of running vanilla jobs for replica registration. Please contact your condor admin to find out if you can. The latest Condor-G release is able to run vanilla jobs.
- The *globus-url-copy* and *transfer* executables requires `GLOBUS_LOCATION` and `LD_LIBRARY_PATH` environment variables to be set in the `tc.data` file. The *replica-catalog* executable entry in the `tc.data` needs the `JAVA_HOME` and `VDS_HOME` environment variables to run correctly. The *rls-client* executable needs all four variables to be set correctly.

13.1 Document Issues to Address

The glossary needs to be revised and sorted into alphabetic order.